OTICE

.pple Computer Inc. reserves the right to make improvements in the
·roduct described in this manual at any time and without notice.

)ISCLAIMER OF ALL WARRANTIES AND LIABILITY

APPLE COMPUTER INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL OR WITH RESPECT TO THE SOFTWARE DESCRIBED IN THIS MANUAL, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. APPLE COMPUTER INC. SOFTWARE IS SOLD OR LICENSED "AS IS". THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAMS PROVE DEFECTIVE FOLLOWING THEIR PURCHASE, THE BUYER (AND NOT APPLE COMPUTER INC., ITS DISTRIBUTOR, OR ITS RETAILER) ASSUMES THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION AND ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES. IN NO EVENT WILL APPLE COMPUTER INC. BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE, EVEN IF APPLE COMPUTER INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

This manual is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer Inc.

© 1980 by APPLE COMPUTER INC.
10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

The word APPLE and the Apple logo are registered trademarks of APPLE COMPUTER INC.

APPLE Product #A2L0041

Apple II

Apple PILOT

Language Reference Manual

ACKNOWLEDGEMENTS

The Apple PILOT Lesson Text Editor incorporates portions of the UCSD Pascal™ Editor. UCSD Pascal was developed largely by the Institute for Information Science at the University of California at San Diego, under the direction of Kenneth L. Bowles.

"UCSD Pascal" is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only and is an indication that the associated product or service has met quality assurance standards prescribed by the University. Any unauthorized use thereof is contrary to the laws of the State of California.

PILOT stands for Programmed Inquiry, Learning or Teaching. The original PILOT was developed in 1968 by Dr. John A. Starkweather and associates at the University of California, San Francisco.

The Apple PILOT programming language is based on a version of the PILOT programming language called COMMON PILOT developed by George Gerhold and Larry Kheriaty at Western Washington University. We thank them for their permission to use material from their Common PILOT Language Reference Manual in this document.

TABLE OF CONTENTS

PREFACE VI

CHAPTER 1
OVERVIEW OF THE LANGUAGE 1

2 What is PILOT?
2 Introduction
5 Correct Form for Instructions
7 The Parts of an Instruction
27 General Information
28 Running a Programmed Lesson

CHAPTER 2
TEXT INSTRUCTIONS 31

32 R: Remark
34 T: Type

CHAPTER 3
RESPONSE INSTRUCTIONS 43

44 PR: PRoblem
48 A: Accept
58 M: Match

CHAPTER 4
CONTROL INSTRUCTIONS 65

66 J: Jump
69 U: Use
74 E: End
77 L: Link
81 XI: Execute Indirect
84 W: Wait

CHAPTER 5

COMPUTATION INSTRUCTIONS 87

88 D: Dimension
93 C: Compute

CHAPTER 6

SPECIAL EFFECTS INSTRUCTIONS 105

106 G: Graphics
119 GX: Execute Graphics File
122 AP: Accept Point
126 TX: Execute Character Set File
129 S: Sound
132 SX: Execute Sound File

CHAPTER 7

FILE HANDLING INSTRUCTIONS 135

136 General File Information
138 FOX: Create and Open New File
142 FIX: Open Existing File
146 FO: Output to Open File
151 FI: Input from Open File

CHAPTER 8

EXECUTION-TIME COMMANDS 155

156 GOTO Command
159 @ Escape Command
162 CTRL-C
163 RESET

CHAPTER 9

HINTS FOR BEGINNERS 165

166 Using the Student's Name
168 Using Numbers
169 Counting Answers
171 An Example

CHAPTER 10

ADVANCED PROGRAMMING 173

174 Constants
175 Variables
188 Functions
201 Operators
206 Expressions

APPENDICES 209

209 Appendix A: ASCII Character Codes
217 Appendix B: Using More Disk Drives
223 Appendix C: Error Messages
233 Appendix D: PILOT Language Summary

INDEX 245

PREFACE

Apple PILOT is a programming language designed for writing teaching programs. Two manuals accompany your Apple PILOT System: the Apple PILOT Editors Manual and the Apple PILOT Language Reference Manual. The Editors Manual explains the four editors you will use to write PILOT programs. The Language Reference Manual tells you how the various instructions in Apple PILOT work. These two manuals are designed for a person who is already familiar with the PILOT programming language but not necessarily familiar with the Apple II computer or the four Editors. You are now reading the Language Reference Manual.

The following is a summary of the contents of each of the two manuals which explain the Apple PILOT System:

Language Reference Manual

Overview of the Language
Text Instructions
Response Instructions
Control Instructions
Computation Instructions
Special Effects Instructions
File Handling Instructions
Execution-Time Commands
Hints for Beginners
Advanced Programming
Appendices

Editors Manual

Overview of the System
Start-Up and Maintenance
Common Editor Features
The Lesson Text Editor
The Graphics Editor
The Character Set Editor
The Sound Effects Editor
Appendices

The capabilities of PILOT and its Editors are demonstrated in a sample PILOT lesson that comes with Apple PILOT. For instructions on using this lesson, refer to the first chapter of the Editors Manual.

Since all Apple PILOT programs are written using the Lesson Text Editor, you should begin learning about the Apple PILOT System by reading chapters 1 through 4 in the Editors Manual. Once you understand the Lesson Text Editor, you can turn to the Language Reference Manual to learn about the instructions which make up the Apple PILOT programming language. For those of you who are not familiar with PILOT, the Introduction in Chapter 1 is a good place to start, followed by Chapter 9's Hints for Beginners. For those who are already very familiar with PILOT, the PILOT Language Summary in Appendix D might be a good starting point.

The rest of Chapter 1 describes in detail all the elements that make up an Apple PILOT instruction. Chapters 2 through 8 tell all there is to know about the operation of each Apple PILOT instruction and execution-time command. Chapter 10, Advanced Programming, reveals facts about the language and its computation abilities that were considered too detailed for the earlier chapters.

As you read the Language Reference Manual you will find that each description of an instruction is followed by a small sample lesson or part of a lesson, showing that instruction in use. These example lessons are carefully annotated to remind you what each instruction is doing. It is recommended that you use your Apple to work your way through each of these examples. You may also want to experiment by creating similar examples of your own. Typing these examples and getting them to run on the computer is an excellent way to learn PILOT.

VIII APPLE PILOT

CHAPTER 1

OVERVIEW OF THE LANGUAGE

2 WHAT IS PILOT?

2 INTRODUCTION

5 CORRECT FORM FOR INSTRUCTIONS

7 THE PARTS OF AN INSTRUCTION
7 Labels
9 Unlabeled Destinations
10 Instruction-Names
11 Modifiers
13 Conditioners
13 Yes and No Conditioners
14 Answer-Count Conditioner
16 Error Conditioner
20 Last-Expression Conditione
21 Expressions
23 The Colon
24 The Continuation-Colon
25 The Object
26 Variables
27 RETURN

27 GENERAL INFORMATION
27 Upper and Lower Case Letters
27 Instruction Length
28 Blank Lines

28 RUNNING A PROGRAMMED LESSON
28 Testing in Author Mode
29 Running in Lesson Mode
29 Execution-Time Commands

WHAT IS PILOT?

Every educator has wished for the ability to provide unlimited, personal attention to each student, in a one-to-one interaction tailored specifically to that student's abilities and needs. Computer-Assisted Instruction (sometimes called CAI) is one attempt to help grant that wish. Using a program written by the instructor, a computer can offer the student individualized tutoring, practice, and testing, with frequent interaction between the student and the program.

Computer-assisted teaching lessons can be written using any interactive programming language, but instructors often know much more about their subject matter than about computer programming. The original PILOT language was developed in the early 1970's in an attempt to solve this problem. The language had only about eight instructions and was designed to let an instructor write teaching programs after just a short time learning the language.

Since that time, PILOT has been revised and extended many times. Today, it is both more complex and much more powerful than the original language. The Apple PILOT programming language is based on the latest standard PILOT development, called COMMON PILOT. COMMON PILOT was developed at Western Washington University; any program written in COMMON PILOT should run correctly on the Apple PILOT system. Apple PILOT includes COMMON PILOT plus special extensions added to take advantage of the Apple computer's graphics and sound capabilities.

INTRODUCTION

The Apple PILOT programming language consists of twenty-two instructions, of which you will use only about nine for writing the largest portions of a lesson. Each instruction consists of two main parts: the instruction itself, and the object of the instruction. The instructions themselves are very simple, but many of them can operate on a wide variety of objects. Furthermore, you can use modifying elements with an instruction to change its operation, and conditioning elements which determine whether or not the instruction will be executed under various conditions. Finally, you can develop more complicated programs and instructional strategies by combining the instructions into complex sequences.

Most instructions are invoked by a single-letter instruction name, followed by a colon. For example, one of the most common instructions is the Type instruction, which causes a question or any other text to be displayed (or "Typed") on the student's screen. The Type instruction is written

```
T:
```


Frequently, the object of the instruction follows the instruction name and colon. In our example, the Type instruction's colon would be followed by the text to be displayed:

```
T: Are you male or female?
```

Most of the remaining common instructions are designed to let the student "talk" to the program. The Accept instruction tells the computer to "listen" while the student types something on the Apple's keyboard:

```
A:
```

The Match instruction then asks the Apple if any word in the Accepted response is the same as the word following the Match instruction name:

```
M: FEMALE
```

If the Apple finds the Match word in the student's response, it stores the answer "Yes"; otherwise it stores the answer "No". This answer can then be used as a condition for executing further instructions. For example, if you want the program to Jump to a program portion labelled WOMEN, dealing with health questions for women, you could use the instruction:

```
J: WOMEN
```

If you want the Jump to occur only when the Apple stores a "Yes" answer to the previous Match instruction, you can add the Yes conditioner to the Jump instruction name:

```
JY: WOMEN
```

This instruction will be skipped if the Apple stores a "No" answer to the previous Match, and the program will go on to the next instruction. This is only one example of the many ways a simple Apple PILOT instruction can be modified for more complex tasks.

An Apple PILOT program is just a sequence of PILOT instructions. Normally, execution of an Apple PILOT program begins with the first instruction and continues by executing the remaining instructions in their order of appearance. However, individual instructions may be skipped if they contain conditions which are not met at that time, and several of the instructions cause execution to jump out of the normal sequence.

Here is a little program that uses the four most common instructions, plus the End instruction which ends a lesson. The comments to the right of the program lines (in parentheses) do not appear in a real lesson; we have added them to explain the action caused by each line.

```
T: Let's talk about your health.      (Display text on screen.)

*GENDER                               (Label for this section.)
T: Are you male or female?            (Display this text.)
A:                                    (Accept student's response.)
M: FEMALE                             (Did student type  FEMALE ?)
JY: WOMEN                             (If Yes, Jump ahead to the
                                        section labelled  WOMEN .)
M: MALE                               (Did student type  MALE ?)
JY: MEN                               (If Yes, Jump ahead to the
                                        section labelled  MEN .)
T: You must be one or the other.      (If the program gets to this
                                        line, student typed neither
                                        FEMALE nor MALE, so display
                                        this text.)
T: Let's try again.                   (Display this text.)
J: GENDER                             (Jump back to the section
                                        labelled  GENDER .)

*MEN                                  (Label for men's section.)
T: OK, you are a man.                 (Display this text.)
T: Now, what is your weight?          (Display this text.)
...
...
E:                                    (End of lesson.)

*WOMEN                                (Label for women's section.)
T: OK, you are a woman.               (Display this text.)
: Now, what is your weight?           (Display this text.)
...
...
E:                                    (End of lesson.)
```

The Apple will store a "Yes" answer to a Match instruction only if the Match word is absolutely identical to a word in the student's last typed response. For instance, if the Match word is FEMALE, the Apple will say "Yes" only if it finds the word FEMALE in the student's response; the response word Female or female or FEMale won't do. For this reason, you can tell the Apple to convert all response letters to capital letters, or all to small letters. See the discussion of the 'Roblem instruction, in the chapter Response Instructions, for details about how to make this happen. Most of the examples in this manual include an instruction which tells the Apple to convert all student responses to capital letters.

Also note that the example shows text for display by the Type instruction in both capital letters and small letters. This is how the text will be displayed to the student, when the lesson is running. However, when you see the program itself, in the Author Mode's Lesson Text Editor, all letters are shown in upper case. True capital letters appear in "inverse video" (black capitals on a white square) while true lower-case letters appear in "normal video" (white capitals on black). All non-letter characters appear correctly in normal video.

On the Apple keyboard, CTRL-Z causes one letter to be shifted to upper case. After a CTRL-Z, the first letter typed will be a capital letter (shown in inverse video), and letters after that will be lower-case (normal video). CTRL-A acts like a typewriter's locking shift key. After CTRL-A, all letters typed will be capital letters, until the next CTRL-A shifts back to lower-case. CTRL-Z and CTRL-A affect letters only; the Apple's SHIFT key works normally for all other characters.

The issue of upper-case or lower-case letters is only important for two instructions: the text to be displayed by the Type instruction, and the text to be compared by the Match instruction. All the rest of your lesson can be in upper-case, in lower-case, or in any mixture; PILOT is indifferent. Since lower-case (displayed in the Lesson Text Editor as white capitals on black background) is often easier to read, you may wish to type most of your lesson in lower-case, saving true upper-case for specific capital letters in the text displayed by Type.

The remaining chapters in this manual discuss the Apple PILOT instructions in detail, with small examples showing their use. In this manual, however, we can only hint at the power of the PILOT programming language. More complete examples are provided in the demonstration programs that you received with your Apple PILOT system. We also urge you to seek out and study examples of PILOT programs from a variety of authors, to learn how the features of the language can be combined. From there, your own imagination will be your guide.

CORRECT FORM FOR INSTRUCTIONS

Like sentences in the English language, instructions in the Apple PILOT programming language must follow a certain form to be understood correctly. In their simplest form, most Apple PILOT instructions consist of only three basic elements: the instruction name, followed by a colon, and then the "object" of the instruction. The object may be text, a calculation, the name of a place to store something, a more detailed command to the Apple, or it may be missing altogether, depending on the instruction. Here is the general format for an Apple PILOT instruction in its simplest form (square brackets indicate that the object is optional):

```
instruction-name:  [object]
```

A common example is the simple form of the Type instruction, which causes the text following the colon to be displayed on the student's screen:

```
T: Hello, what is your name?
```

The longest line you can see in the Lesson Text Editor is 40 characters, since that is the width of the screen. To make longer instructions, Apple PILOT lets you "continue" an instruction on subsequent lines (up to 250 characters in all), by starting each continuation line with another colon:

```
T:Four score and seven years ago, our
:forefathers brought forth upon this
:continent a new nation, conceived in
:liberty and dedicated to the
:proposition that all men are created
:equal.
```

Any PILOT instruction can be extended this way. When the instruction is executed, the Apple just ignores the breaks and extra colons, treating the instruction as one long, unbroken line.

There are three ways to alter the execution of a particular instruction. Each requires the addition of another element after the instruction name and before the colon. These three elements are called "modifiers", "conditioners", and "expressions".

A "modifier" changes some detail of how an instruction works. Most modifiers work on only one or two instructions. For example, the Type instruction normally puts its object text on the screen, and then jumps to the beginning of the next line. If the student then types a response, it appears on that next line. But a Type instruction with the Hang modifier stops ("Hangs") at the end of the displayed text, so the student's response will appear right there, instead of on the next line:

```
TH:  7 + 13 =
```

A "conditioner" causes an instruction to be executed or skipped, depending on a particular internal test. Conditioners can be used with almost all instructions. An instruction used with the Yes conditioner, for example, will be executed only if the previous Match word was found in the student's typed response:

```
TY: Good, that is the correct answer.
```

An "expression" is similar to a conditioner, except that you define your own test. The expression performs a calculation or comparison whose result causes the instruction to be executed or skipped. An instruction used with the expression (N>13) , for example, will be executed only if the number-storage place named N contains a number which is greater than thirteen at that moment:

```
T(N>13): No, your answer is too large.
```

In addition to these three kinds of instruction-changers, an element called a "label", preceded by an asterisk, can appear alone or before an instruction name. A label identifies the beginning of a program section, and can be used as a destination by instructions which need to jump to that section:

```
*TEST   T: OK, let's try some problems.
```

A little later in this chapter, you will find a detailed discussion of each possible element in an instruction. When more than one of these

elements is used in an instruction, the elements must appear in a certain order. Here is the general format showing the correct order for all the possible elements of an Apple PILOT instruction (square brackets surround optional elements):

```
[*label] instruction-name[modifiers&conditioners][(expression)]: [object]
```

An instruction need not contain all the possible elements, but any elements included must be in the order specified. Here is another way to see the order of possible elements in an instruction:

Order	Element	Comment
1.	*label	Optional; may be on separate line.
2.	instruction-name	Required; one, two, or three letters.
3.	modifiers	Optional; usually just one; one letter.
	conditioners	Optional; 4 allowed; one character each.
4.	(expression)	Optional; one allowed; in parentheses.
5.	:	Required colon.
6.	object	Depends on the instruction.
7.	RETURN	Required at end of every instruction.

Note: in standard PILOT, only one modifier is allowed, and it must precede the conditioners. In Common PILOT and in Apple PILOT, any number of modifiers and conditioners may appear in any order, as long as they follow the instruction name and precede the expression (if any) and the colon.

Also note: instructions which do not follow the correct form may be executed anyway, but with unpredictable results.

In the following section, the elements of an Apple PILOT instruction are discussed in detail.

THE PARTS OF AN INSTRUCTION

LABELS

A label is a word that identifies an instruction or program section so that other instructions in the lesson can refer to that place. A label is an optional element which may precede any Apple PILOT instruction. Not all lines need to be labelled, and labels can stand alone on a line (although such a line is not considered to be an instruction).

When the label is placed in a program, it must begin a new line. An asterisk (*) as the first character of a line indicates to PILOT that the element which follows is a label. The label itself is a name up to six characters in length. (A label consisting of no characters at all is valid, but obviously has no section-identifying quality whatsoever. Its use is discouraged.) Any printing character may be used for any of the characters in a label name, but a label must not contain spaces in the middle of the name or before the name. It does

not matter whether you type capital letters or small letters (or any mixture), nor do you have to be consistent in that regard from one use of the label name to the next. If the label is on the same line as an instruction, the label must be separated from the instruction name by at least one space.

For example, you can add a label before the instruction name of a Type instruction in either of these two equivalent ways:

```
*START  T: Hello, what is your name?
```

or

```
*START
T: Hello, what is your name?
```

Other instructions can cause program execution to branch to (or "go to") the labelled instruction. When a label name appears in the object field of an instruction to specify the destination for a branching operation, an asterisk does NOT precede the label name. For instance, you might wish the program to Jump to the instruction labelled in the example above:

```
J: START
```

The instruction `J: start` would also do.

Label names can also be used by both author and student, when the lesson is running, to choose certain sections for testing or review and to skip other sections. One label name has a special meaning in this context: if the Escape option has been enabled by the last PRoblem instruction, any response beginning with the character @ will cause an automatic branch to the label SYSX . See the chapter Execution-Time Commands for details.

To avoid ambiguity, a particular label name should be placed at only one location within a lesson (but see the Wipe-labels option of the PRoblem instruction, in the chapter Response Instructions, for exceptional cases). You will find it helpful to insert labels in your program according to a coherent lesson plan, thought out in advance. This will make the program more readable, helping you and others to understand and modify your program in the future. The following is an example of a possible labelling scheme:

```
*START
...
...
*INTRO
...
...
*PART1
...
...
*PART2
...
...
*PART3
...
...
*REVIEW
...
...
*TEST
...
...
*END
```

Apple PILOT keeps up to 5Ø of a lesson's labels in a reference table, so program execution can jump to those places in the lesson quickly. When jumping to a label that has not yet been put in the table, or to a label that did not fit in the table, Apple PILOT searches through the lesson until it finds the label. In a large lesson, this can be fairly slow, so try to limit the number of labels you use to fewer than 5Ø .

Using unlabelled destinations, discussed in the next sub-section, will help reduce the number of labels you need. See the Wipe-labels option of the PRoblem instruction, in the chapter Response Instructions, for details on starting a new label table in the middle of a lesson.

Unlabelled Destinations

In addition to the instructions that you have labelled in your lessons, certain instructions can be used as destinations for branching operations, even though they are not explicitly labelled. To specify a branch to one of these instructions, the following notation is used:

Notation	Branch Destination Specified
@A or @a	the last Accept instruction previously executed
@M or @m	the next Match instruction
@P or @p	the next PRoblem instruction

These unlabelled relative branch destinations can be used with the instructions Jump, Use, and End, as well as by the GOTO execution-time command (if enabled).

Using the unlabelled destinations helps to reduce the number of labels your lesson needs. Up to 5Ø labels can be used without much slowing down of your lesson, so this is rarely a limitation. However, these destinations can be very useful for other purposes, too. For example, if each independent section of a lesson begins with a PRoblem instruction, instructions in any section can branch to the next section just by instructing:

```
J: @P
```

With this system, you can re-arrange the sections, remove sections, or add new ones, and this branch will still work. If you were to remove a section identified by a specific label, all instructions which branched to that section by its label name would have to be changed.

Another common use is to write a lesson section which handles any error in a student's typed response. After handling the error, perhaps just by telling the student to "Try again" or "Type a number", the program can jump back to the unlabelled destination @A . This error-handling section can be used by every Accept instruction in a lesson, and it will always return to the last Accept instruction which accepted a response.

INSTRUCTION NAMES

The instruction name is a required element; it specifies the particular kind of action to take when the Apple executes that line in the program.

There are only two kinds of lines that do not include an instruction name: the line which consists of a stand-alone label, and a line which starts with a colon. When an instruction (such as Type or Remark) is continued beyond a single line, the subsequent lines of the same instruction are started with a colon. In this case, the colon implies that the new line is really still part of the previous instruction line. When PILOT executes the instruction, it ignores any such breaks and colons in the instruction line, treating the whole instruction as one long line.

The instruction name usually consists of just one or two letters, which are usually an abbreviation for the action specified. For example, the instruction name for the Type instruction, which displays or "types" information on the screen, is T .

The instruction name must be separated from any preceding label on the same line by at least one space, so that PILOT can tell where the label ends and the instruction name begins. If there is no preceding label, the instruction name begins the line. There may be any number of spaces between the instruction name and the element that follows it (which may be a modifier, conditioner, expression, or colon).

Here are the Apple PILOT instruction names, presented in the order in which they are discussed in the chapters following this one.

Chapter Title	Name	Full Name	Description
Text Instructions	R	Remark	Remarks to the author; not executed.
	T	Type	Displays text on the student's screen.
Response Instructions	PR	PRoblem	Controls the handling of typed responses.
	A	Accept	Accepts the student's typed response.
	M	Match	Determines if student typed certain words.
Control Instructions	J	Jump	Starts doing another part of the lesson.
	U	Use	Like Jump, except it "remembers" how to return here when the other part is done.
	E	End	Ends a lesson or part.
	L	Link	Starts a new lesson.
	XI	eXecute Indirect	Executes stored words as an instruction.
	W	Wait	Waits for a set time.
Computation Instructions	D	Dimension	Reserves spaces for storing words or groups of numbers.
	C	Compute	Does calculations and stores the results.
Special Effects Instructions	G	Graphics	Draws lines and dots on the screen.
	GX	eXecute Graphics	Draws a diskette-stored image on the screen.
	S	Sound	Plays musical notes on Apple's speaker.
	SX	eXecute Sound	Plays a diskette-stored sound effect.
	TX	eXecute Characters	Changes the character s used for text display
File Handling Instructions	FOX	Open New File	Creates and opens a new diskette file.
	FIX	Open Old File	Opens an existing file on the diskette.
	FO	File Output	Stores information in the open file.
	FI	File Input	Retrieves information from the open file.

MODIFIERS

A modifier is a one-letter optional element which changes some detail of the way an instruction works. Any number of modifiers, in any order may be used with any particular instruction, but it rarely makes sense to use more than one modifier. The preceding element may be the instruction name, another modifier, or a conditioner. The element which follows may be another modifier, a conditioner, an expression, or a colon.

A modifier usually invokes a feature designed for just one instruction, and invokes a different feature or is ignored if it is used with other instruction names. For example, after a normal Type instruction displays its text, the cursor automatically jumps to the beginning of

the next line. To make the cursor "hang" at the end of the displayed text (so the student's response, for instance, will appear on the same line) you can add the H modifier to the Type instruction name:

```
TH:  5 + 13 =
```

The H modifier has no effect when used with any other instruction.

Apple PILOT has seven special modifiers for use with its instructions. These modifiers work only with the four instructions listed, unlike the conditioners (see next section) which are effective on all instructions. The modifiers, appended to the instruction names of the instructions they affect, are:

TH:	(Hang modifies Type)	Stops carriage-return after a Type.
AX:	(eXact modifies Accept)	Accepts exact response, with no editing.
AS:	(Single modifies Accept)	Accepts a single-character response.
AP:	(Point modifies Accept)	Accepts an X,Y point from game controls.
MS:	(Spell modifies Match)	Allows one-letter misspelling on a Match.
MJ:	(Jump modifies Match)	Jumps to the next Match if this Match is unsuccessful.
LX:	(Erase modifies Link)	Starts a new lesson without preserving the old lesson's variables.

In a few cases, you can use more than one modifier sensibly. For instance, the instruction:

```
MSJ: APPLE
```

looks for the word APPLE in the student's response, allowing certain kinds of misspelling, and also jumps ahead to the next Match instruction if the Match word is not found in the response. However, the nonsense instruction

```
ASPX:
```

functions just like an AP: instruction.

In addition to the seven modifiers discussed above, there is an X (for eXecute) modifier which can be used with five more instructions. Technically, these combinations would each be considered modified instructions, because the instruction can be executed without its modifier. However, the X modifier executes additional parts of the Apple PILOT language, which change the effect of the associated instructions so much that these modifications are usually discussed as separate instructions:

TX:	(X modifies Type)	Changes character set used for display.
SX:	(X modifies Sound)	Plays sound-effect stored on diskette.
GX:	(X modifies Graphics)	Draws image stored on diskette.
FIX:	(X modifies File Input)	Opens an existing diskette file.
FOX:	(X modifies File Output)	Creates and opens new diskette file.

For examples and more details about the various modifiers, see the discussions of the instructions affected by each individual modifier, in the chapters following this one.

CONDITIONERS

A conditioner is an optional element; it establishes a condition which, if not met, causes execution to skip to the following instruction. Each conditioner is a single letter or a single numeric digit.

Any number of conditioners, in any combination and in any order, may be used in an Apple PILOT instruction. However, a maximum of four conditioners (one each of the four different kinds) may be usefully employed in one instruction. Where more than one conditioner is used in an instruction, the instruction will be executed only if all of the specified conditions are met at execution time. Unlike modifiers, which work only with certain instructions, any conditioner can be used effectively with any Apple PILOT instruction.

The instruction name must immediately precede the list of modifiers and conditioners, with no intervening spaces.

There are four different kinds of conditioners, each of which can determine under what conditions an instruction will be executed:

As Typed	Name	Condition Established
Y	(Yes conditioner)	Execute if last Match was successful.
N	(No conditioner)	Execute if last Match was unsuccessful.
1 to 9	(Answer-Count conditioner)	Execute if the conditioner number equals the number in the Answer Counter, which counts how many times the student has tried to answer the same question.
E	(Error conditioner)	Execute if the Error Flag has been raised by trouble in executing any instruction.
C	(Last-Expression conditioner)	Execute if the last evaluated instruction-modifying expression was True (non-zero).

In the following sections, each of these conditioners is discussed in more detail.

Yes and No Conditioners

The Yes conditioner is always a Y , and the No conditioner is always an N . These conditioners test the success of the last Match instruction executed. The Y conditioner allows an instruction to be executed only if the previous Match was successful (a "Yes" result); that is, the Match word was found somewhere in the student's response. The N conditioner allows an instruction to be executed only if the previous Match was not successful (a "No" result). It does not make sense to use both a Yes and a No conditioner in the same instruction, but if you do

this anyway the instruction is executed as if it had only the Yes conditioner.

These conditioners help form the basic interaction between an Apple PILOT lesson and the student. They let the program respond differently, depending on whether a student does, or does not, give an expected response.

Example:

```
*TEST
T: What is the highest
: mountain in the world?
A:
M: EVEREST
TY: That's right.
TN: No, try again.
JN: TEST
```

(Label for this section.)
(Display this text, and this text.)
(Accept student's response.)
(Did student type EVEREST ?)
(If Yes, display this text.)
(If No, display this text.)
(If No, Jump back to TEST .)

Example:

```
*DAY
T: Is today sunny, or cloudy?
A:
M: SUNNY
TY: How nice!
JY: NEXT
M: CLOUDY
TY: Too bad.
TN: I don't understand.
: Please try again.
JN: DAY
*NEXT
```

(Label for this section.)
(Display this text.)
(Accept student's response.)
(Did student type SUNNY ?)
(If Yes, display this text.)
(If Yes, Jump to label NEXT .)
(Did student type CLOUDY ?)
(If Yes, display this text.)
(If No, display this text and also this text.)
(If No, Jump back to DAY .)
(Label for next section.)

Answer Count Conditioner

his conditioner is always a single numeric digit, from 1 through 9 . t tests the current value of the Answer Counter. The Answer Counter ndicates how many times in a row the most recent Accept instruction has een encountered, with no other Accept instruction intervening. The nswer-Count conditioner allows an instruction to be executed only if he conditioner number is the same as the current value of the Answer ounter. If more than one numeric digit appears in the list of modifiers nd conditioners for an instruction, only the last (rightmost) digit is sed. For more information about the Answer Counter, see the Accept nstruction, in the chapter Response Instructions.

his conditioner lets the program respond differently, depending on how ny times the student has attempted to answer the same question. On e third try, for instance, you might want to provide the answer to e question, or you might branch to a review section. Without this nditioner, as the examples in the previous section show, the student uld be stuck on one question indefinitely.

Example:

```
*TEST
T: What is the highest
:mountain in the world?
A:
M: EVEREST
TY: That's right.
TY1: You got it the first try!

TN4: No, Mount Everest is the
:world's highest mountain.
JN4: NEXT

TN: No, try again.
JN: TEST
*NEXT
```

(Label for this section.)
(Display this text, and this text.)
(Accept student's response.)
(Did student type EVEREST ?)
(If Yes, display this text.)
(If Yes and first answer, displny this text.)
(If No and fourth answer, display this text .)
(If No and fourth answer, Jump to label NEXT .)
(If No, display this text.)
(If No, Jump back to TEST .)
(Label for next section.)

In the example above, the student will be told to "try again" after the first three wrong answers. If the student's fourth answer is still wrong, the program shows the correct answer and goes on to the next question (*NEXT).

Example:

```
*DUKBIL
T: What mammal has a bill,
:and lays eggs?
A:
M: PLATYPUS
TY1: Right, on the first try!

TY2: Right, on the second try.

JY: KOALA
T1: No, try again. Hint:
:it lives in Australia.
J1: DUKBIL

T: I think you should
:review the following.
J: REVIEW
*KOALA
```

(Label for this section.)
(Display this text, and this text.)
(Accept student's response.)
(Did student type PLATYPUS ?)
(If Yes and answer #1, display this text.)
(If Yes and answer #2, display this text.)
(If Yes, Jump to *KOALA .)
(If wrong answer #1, display this text.)
(If wrong answer #1, Jump back to label TEST .)
(Second wrong answer: display this text, and Jump to label REVIEW .)
(Label for next section.)

The Answer Count is kept in the "system variable" (one of Apple PILOT's number-storage places) named %A . For a different kind of example, which tests the %A Answer Count in an expression, see the section Expressions later in this chapter.

Error Conditioner

This conditioner is always an E . It tests the current state of the Error Flag. The Error Flag is an indicator in the Apple that is "raised" any time Apple PILOT has trouble executing one of your instructions. The error conditioner allows an instruction to be executed only if the Error Flag is currently raised. This lets you write instructions designed to solve certain predictable errors that may arise when the lesson is running.

When you run a lesson from the Lesson Text Editor, in Author Mode, most problems with instructions cause error messages to be displayed on the screen (a section called Running a Programmed Lesson, later in this chapter, discusses error messages). These problems, which may be the result of a mis-typed instruction, a division by zero in a calculation, etc., also raise the Error Flag at the same time. When the student is running a lesson in Lesson Mode, however, no error messages are displayed: the Error Flag being raised is the only indication that an instruction has run into difficulty.

In addition to the many kinds of problems that can display a message (in Author Mode only) while raising the Error Flag (in either mode), there are two situations which raise the Error Flag without causing an error message in either Author Mode or Lesson Mode:

1. If an Accept instruction is supposed to store a student's typed number in a specified number-storage place, but no number is found in the student's response, no new number is stored in the number-storage place, and the Error Flag is raised. For example, this instruction:

```
A: #N                    (Accept the student's response
                          and store the first typed
                          number in the number-storage
                          place named N .)
```

   will raise the Error Flag if the student types only letters, or presses the RETURN key without typing anything.

2. If a FIX: (open an existing diskette file) instruction does not find a file with the specified name already stored on the diskette, no file is opened, and the Error Flag is raised. For example, this instruction:

```
FIX: 1Ø,S$               (Find an existing diskette file
                          whose name is the word stored
                          in the word-storage place named
                          S$ , and open it for storing
                          and retrieving information in
                          records numbered Ø through 1Ø.)
```

   will raise the Error Flag if it finds no file with the given file name.

The Error Flag is always down when a lesson starts, and stays down until something raises the flag. Once the Error Flag has been raised, it stays up indefinitely unless something lowers it. There are many kinds of problems which can cause the Error Flag to be raised, but only five things can lower the Error Flag while a lesson is running:

1. The Error Flag is lowered just before evaluating any mathematical or string expression. Thus, successfully evaluating any execution-modifying expression (discussed later in this chapter) or any Compute instruction's expression lowers the Error Flag. For example, either of these instructions will lower the Error Flag if it is up:

```
T(N=12):That's right.    (If the execution-modifying
                          expression (N=12) is true,
                          display this text.)

C: X = 5+3               (Evaluate the expression 5+3
                          and store the result in the
                          number-storage place named X .)
```

   However, this instruction leaves the Error Flag raised:

```
C: X = 5/Ø               (The illegal expression 5/Ø
                          causes an error message in
                          Author Mode and raises the
                          Error Flag in either mode.)
```

2. The Error Flag is lowered just before executing the contents of the word-storage place whose name appears in the object field of an XI: instruction. Thus, successfully executing an XI: instruction lowers the Error Flag. For example, this instruction will lower the Error Flag if the word-storage place named A$ contains a correct, executable PILOT instruction:

```
XI: A$                   (Execute the string of characters
                          stored in the word-storage place
                          named A$ as a PILOT instruction.)
```

3. If an Accept instruction successfully stores a student's typed number in a specified number-storage place, the Error Flag is lowered. For example, this instruction will lower the Error Flag if the student's typed response includes a number:

```
A: #N                    (Accept the student's response,
                          and store the response's first
                          number in the number-storage
                          place named N .)
```

4. Successfully executing any PRoblem instruction lowers the Error Flag. For example:

```
PR:                          (Mark a new section without
                               changing any options.)
```

5. Any LX: instruction, which starts a new lesson running, forgetting everything about the old lesson, lowers the Error Flag. For example:

```
LX: LESSON2                  (Start running the lesson named
                               LESSON2, without keeping old
                               stored words and numbers.)
```

Flag-lowering item number 1 provides a way to lower the Error Flag intentionally whenever you want to do so. Ordinarily, you will want to lower the Error Flag at the end of any lesson section designed to handle a particular error condition. This prevents later E conditioned instructions from responding to the old error which you have already handled. Any legal expression in an instruction will lower the Error Flag, including a simple expression such as (1) . For example:

```
T:How old are you?           (Display this text.)
A: #N                        (Accept student's response;
                               store number in number-
                               storage place named N .)
TE:Please type a number.     (If Error Flag is up, student
                               did not type any number, so
                               display this text.)
JE(1): @A                    (If Error Flag is up, lower
                               Error Flag and Jump back to
                               last Accept instruction.)
```

The E conditioner is usually used in conjunction with the two special flag-raising situations that do not cause error messages in Author Mode. Used after situation 1, the Error conditioner lets the program correct for an invalid (in this case, non-numeric) student response. Used after situation 2, the Error conditioner lets the program determine whether a student has already saved some information under a given file name, and acts accordingly.

Most of the other problems that raise the Error Flag can be traced to incorrectly typed instructions or other errors which you clear up in the testing process, while running the lesson from Author Mode.

Example:

```
*SUM1                        (Label for this section.)
T: How much is  7 + 5  ?     (Display this text.)
A: #N                        (Accept student's response,
                               and store the first number
                               in number-storage place N .)
TE: Please type a number.    (If the Error Flag is up, no
                               number was typed, so display
                               this text.)
JE(1): @A                    (If the Error Flag is up, lower
                               it and Jump back to the last
                               Accept instruction.)
M: 12                        (Did student type 12 ?)
TY: Yes, 12 is correct.      (If Yes, display this text.)
JY: SUM2                     (If Yes, Jump ahead to SUM2 .)
T: No, try again.            (Wrong answer: display text,
J: SUM1                        and Jump back to SUM1 .)

*SUM2                        (Label for next section.)
```

This example forces the student to type a number, rather than spelling it out. See the Match instruction, in the chapter Response Instructions, for alternate solutions that would allow either form of response.

Note: you would not normally try to Match a student's numeric response. The instruction M:12 will successfully Match responses such as 12, 412Ø, or even 3.2E12 . The example at the end of the section on Expressions, later in this chapter, shows a better way to test numeric responses.

Example:

```
D: Q$(1Ø)                    (Create a word-storage place
                               named Q$ , for storing
                               words up to 1Ø characters
                               long.)
T: Please type your first name.  (Display this text.)
A: $Q$                       (Accept student's response and
                               store it in the word-storage
                               place named Q$ .)
FIX: 9,Q$                    (Try to open an existing disk
                               file whose name is the name
                               stored in Q$ , for storing
                               and retrieving records Ø
                               through 9 .)
FOXE(1): 9,Q$                (If the Error Flag is up, no
                               old file was found that had
                               the same name as the student,
                               so create and open a new
                               1Ø-record disk file with the
                               name stored in Q$ , after
                               lowering the Error Flag.)
*MAIN                        (Label for main lesson.)
```

The example above attempts to open a previously-stored file named after the student. If no previous file by that name is found, a new file by that name is created. This file could be used for storing the student's scores, specific responses, or other information.

Last-Expression Conditioner

This conditioner is always a C (do not confuse this with the name of the Compute instruction, which is also a C). It tests whether the last evaluated instruction-modifying expression was True (non-zero) or False (zero).

An instruction-modifying expression is a calculation or comparison which may appear in parentheses immediately after an instruction's list of conditioners and modifiers. The result of evaluating that expression is used as another test to decide if that instruction will be executed (see the next section in this chapter for details).

The last-expression conditioner C can be used with instructions subsequent to the instruction which contains an instruction-modifying expression. An instruction using the C conditioner will be executed only if the most recently evaluated instruction-modifying expression (in a previous instruction) was True (non-zero). See the section Expressions next in this chapter, and the chapter Advanced Programming for more examples and details of expressions.

The use of the last-expression conditioner simplifies the use of expressions as instruction modifiers. Where the same instruction-modifying expression applies to several instructions in a series, the C conditioner can be substituted for the expression in all the instructions after the first one. Only the original expression is actually evaluated; the result of that evaluation is then applied automatically to the subsequent C conditioned instructions.

Note: the C conditioner refers to the last instruction-modifying expression which was actually evaluated. If an instruction is not executed (because of a conditioner or a previous Jump instruction, for instance), any instruction-modifying expression in that instruction is NOT EVALUATED. Therefore, a subsequent C conditioner will NOT refer to that instruction's expression.

Example:

```
*STATE                                  (Label for this section.)
T: Which state has the longest days?    (Display this text.)
M: ALASKA                               (Did student type ALASKA ?)
TY: Yes, that's right.                  (If Yes, display this text.)
JY: NEXT                                (If Yes, Jump to label NEXT .)
M(%A<3):WASHINGTON!IDAHO!MONTANA        (If the Answer Count stored in
:!DAKOTA!MINNESOTA                        the number-storage place named
                                          %A is less than 3, did student
                                          type any of these words?)
TYC: Good guess, but it's               (If Yes and if Answer Count %A
:still farther north.                     is less than 3, display text.)
TNC: No, it's farther north.            (If No and if Answer Count %A
                                          is less than 3, display text.)
TC: Try again.                          (If Answer Count %A is less
                                          than 3, display this text.)
JC: STATE                               (If Answer Count is less than
                                          3, Jump back to label STATE .)
T: No, Alaska has the longest days.     (3rd bad answer: display text
                                          and go on to next section.)
*NEXT                                   (Label for next section.)
T: Some days the sun does not set       (Display this text.)
:at all, in northern Alaska.
```

The example above tells the student to "try again" after the first two wrong answers, and provides the correct answer after the third wrong answer. Remember that %A is the name of the number-storage place where the Apple stores the current Answer Count. In the second Match instruction, the "or" symbol (!) separates multiple Match words (see the Match instruction for details).

EXPRESSIONS

An instruction-modifying expression is an optional element. It perform a calculation or makes an "assertion", and causes the instruction to be executed or to be to skipped, depending on the result of the calculatio or the truth of the assertion. If the expression, when evaluated, is non-zero or true, the instruction may be executed. If the evaluated expression is zero or false, the instruction is skipped.

The instruction-modifying expression is a single mathematical calculation or a single assertion, enclosed in parentheses. Only one expression may modify an instruction, but that expression can contain any valid combination of sub-expressions. You will have to read the chapter Advanced Programming to discover all the rules and all the powers of expressions. Any expression can be used with any Apple PILOT instruction.

The instruction name and its list of modifiers and conditioners (if any must immediately precede the expression. The colon must immediately follow the expression. There must be no space immediately preceding th expression's opening parenthesis, and there must be no space between th expression's closing parenthesis and the following colon. The expressi may contain internal spaces.

Any conditioners used must precede the expression and are tested first. If any conditioner causes the instruction to be skipped, the expression is NOT EVALUATED. This is normally important only when subsequent instructions use the C (last-expression) conditioner, which refers to the last instruction-modifying expression actually evaluated.

Any expression in Apple PILOT lowers the Error Flag if the expression is successfully evaluated (that is, no errors are encountered in the evaluation). Even a simple expression such as (1) or (Ø) can be used to lower the Error Flag. If the expression contains an error, its evaluation will raise the Error Flag.

An "assertion" expression is a statement which, if true, allows the instruction to be executed. If the assertion is false, the instruction is skipped. Here are some sample assertion expressions:

Expression	Assertion
(5<2)	"5 is less than 2"
(N > 4)	"The number now stored in the number-storage place named N is greater than 4 ."
(A&B)	"The number stored in number-storage place A is not zero and the number stored in B is also not zero."
((%A<4) & (X=Y))	"The assertion 'the number in %A is less than 4 ' is true and the assertion 'the number in X equals the number in Y ' is also true."

Example:

```
T(X>14): That's too large          (If the assertion "the number
                                     stored in number-storage
                                     place named X is greater
                                     than 14" is true, display
                                     this text.)
```

A "calculation" expression is eventually reduced to a number. If that number is not zero, the instruction is executed. If the number is zero, the instruction is skipped. Here are some sample calculation expressions:

Expression	Resulting Number
(7)	7
(12-(2*6))	ø
(Q-13)	13 subtracted from the number now stored in the number-storage place named Q .
((3*A) - (4/B))	4 divided by the number in B , subtracted from 3 times the number stored in A .

Example:

```
T(G+5): No, the answer is -5 .     (If the result of adding 5 to
                                     the number stored in number-
                                     storage place named D is not
                                     zero, display this text.)
```

The rules for writing expressions are discussed fully in the chapter Advanced Programming.

Example:

```
*EXAM                              (Label for this section.)
T: How much is  5 plus 7  ?        (Display this text.)
A: #N                              (Accept student's response, and
                                     store the response's first
                                     number in storage-place N .)
TE: Please type a number.          (If Error Flag is up, no number
                                     was typed, so display text.)
JE(1): @A                          (If Error Flag is up, lower it
                                     by evaluating the non-zero
                                     expression (1) and Jump
                                     back to the last Accept
                                     instruction executed.)
T(N=12): Yes,  12  is correct.     (If the assertion "the number
                                     in N equals 12" is true,
                                     display this text.)
JC: NEXT                           (If the assertion "the number
                                     in N equals 12" is true,
                                     Jump to label NEXT .)
T(N>12): No, #N  is too large.     (If the assertion "the number
                                     in N is greater than 12" is
                                     true, display this text,
                                     substituting the number in N
                                     for #N and one space.)
T(N<12): No, #N  is too little.    (If the assertion "the number
                                     in N is less than 12" is
                                     true, display this text,
                                     substituting the number in N
                                     for #N and one space.)
T(%A<3): Please, try again.        (If assertion "the Answer Count
                                     in %A is less than 3" is
                                     true, display this text.)
JC: EXAM                           (If assertion "the Answer Count
                                     in %A is less than 3" is
                                     true, Jump back to EXAM .)
T: You need some review.           (3rd wrong answer: display text
J: REVIEW                            and Jump to REVIEW .)

*NEXT                              (Label for next section.)
```

When the two middle Type instructions are executed, PILOT replaces the symbol #N , including the first space after, with the number stored in the number-storage place named N . See the Type instruction, in the chapter Text Instructions, for details.

THE COLON

Every instruction line must contain a colon (:). It is usually preceded by the instruction name, list of modifiers and conditioners (if any), and expression (if any), with no intervening spaces. Most instructions ignore spaces between the colon and the object (if there is an object), but the Type instruction will display those spaces.

The Continuation-Colon

If the colon is the first non-space character on a line, the line is considered a continuation of the previous instruction, and also continues that instruction's modifier, conditioners, and expression, if any.

Note: do not allow a blank line (two RETURN characters in a row) to separate a continuation of the instruction from the previous portion of that instruction.

The Lesson Text Editor automatically breaks up any instruction that exceeds 39 characters as you type it in, and continues it (beginning with the continuation-colon) on the next line. You can continue any instruction, or add another piece in the middle, by adding lines that begin with a continuation-colon. When the Apple executes an instruction continued on extra lines beyond the line starting with the instruction's name, it ignores the breaks and the continuation-colons, treating the instruction as if it had been written on a single, long line.

Only the first 25Ø characters of an instruction are used when that instruction is executed. This is about six full lines of text. The continuation-colons are not counted in the 25Ø-character limit, but all other characters are counted. If any portion of an instruction extends beyond the 25Ø-character limit, that portion of the instruction is ignored.

Remark instructions can be continued indefinitely, as they are not executed.

Example:

```
R: THE FOLLOWING EXAMPLES
: SHOW SOME OF THE USES
: AND EFFECTS OF THE
: CONTINUATION-COLON.
```
(Remark instruction is not executed, so it can be continued indefinitely.)

```
TY2:Now is the thyme for all
:good beadles to come to
:the aid of their crumhorn,
:and lilacs to graze on the
:summer's daze, and bats to
:their belfries fly, come morn.
```
(If conditioners allow this instruction to be executed, the words in all these lines of text will be re-arranged into a solid paragraph 4Ø characters wide on the student's screen. Type treats extended text as one long line, ignoring the continuation-colons; 25Ø characters maximum.)

```
M: MERCURY|VENUS|EARTH|M
:ARS|JUPITER|SATURN|URAN
:US|NEPTUNE|PLUTO
```
(Match treats extended list as one long line, ignoring the continuation-colons; 25Ø characters maximum.)

```
G: VØ,39,Ø,1Ø;ESØ;C6;P2Ø,2
:Ø;D54Ø,2Ø;D54Ø,2ØØ;D2Ø,2Ø
:Ø;D2Ø,2Ø;M2ØØ,4Ø;C9;D2ØØ,
:21;P3Ø,3Ø;C5;P4Ø,4Ø
```
(Graphics treats extended list as one long line, ignoring the continuation-colons; 25Ø characters maximum.)

```
S: 2,12;2,12;8,48;2,12;8,
:12;12,48;2,12;8,12;12,12
:;2,12;8,12;12,12;2,12;8,
:12;12,48
```
(Sound treats an extended list as one long line, ignoring the continuation-colons; 25Ø characters maximum.)

The examples above have been formatted somewhat arbitrarily, to fit on this page with our comments. The lesson Text Editor will automatically break your instructions as you type them, to stay within 39 character spaces.

THE OBJECT

Not every instruction has an object, and the object is often an optional element. It must follow the instruction's colon. Most instructions ignore spaces between the colon and the object, but Type will display all such spaces.

The object may be text, a number, the name of a word-storage place, the name of a number-storage place, a diskette file name, the name of a place in the lesson to jump to, a list of more detailed commands to carry out, control-options which set how things work in PILOT, etc., depending on the instruction.

Names of Instructions	Contents of Object	Examples in Instructions:
Remark, Type	Text	T: What is your age?
Match		M: ALASKA|HAWAII
Jump, Use, End	Branch destination	J: @A E: END
PRoblem	Control options	PR: L G T1Ø
Graphics	Command list	G: ES1;C2;M8Ø,9Ø;D48Ø,9Ø
Sound		S: 2,12;2Ø,1ØØ
Wait	Number	W: 5
Accept	Name of word or number store-place	A: #N S
eXecute Indirect		XI: Q$
GX:, SX:, TX:, L:	File name	GX: PICTURE3

Names of Instructions	Contents of Object	Examples in Instructions:
Type	Mixtures	T: Yes, S , #N is right.
Dimension	of different	D: Q$(5Ø)
FIX:,FOX:,FI:,FO:	kinds of	FIX: 2Ø,X$
Compute	elements	C: N = 3*(17/149.6)

Variables

Very often in this manual you will see the word "variable" used. This word is just a special term for a number-storage place or a word-storage place in the Apple. They are called variables because you can change, or vary, the numbers or words in them whenever you wish.

Variables are very useful. You give one a name, store a number or word in it, and from then on you can use the name of that variable in place of the number or word stored there. If you store a different item in that variable, the new item is used from then on, everywhere that variable's name appears.

The names you give to variables are always one letter, or one letter followed by an integer from Ø through 9. The name of a word-storage place is always followed by a dollar sign ($). Up to 5Ø different variable names may be used in a lesson.

Several different kinds of variables may be used. A variable which stores just one number is called a "simple numeric variable". That is the most common kind of number-storage place. It has a name such as N , or X , or T3 . You can create one by storing a number in it with a Compute instruction such as C: T3 = 5 or with an Accept instruction such as A: #T3 .

A variable which stores a word (or any collection or "string" of characters) is called a "string variable". It has a name such as G$, or S$, or R5$. You use a Dimension instruction such as D: S$(3Ø) to create a string variable and reserve enough space for the strings you expect to store there. It is also possible to use parts of a string variable; these parts are called "substring variables" (see the chapter Advanced Programming).

For some purposes, it may be handy to have a single name refer to a contiguous sequence of number-storage places. This is called a "numeric array variable". It has a name such as Y , or P , or W2 . A "subscript" number (or pair of numbers) can be attached to the name of a numeric array variable to indicate which of the number-storage places in that set you need at the moment. These individual "elements" of the array variable have names such as Y(3) , or P(4,5) , or W2(13) . You use a Dimension instruction such as D: Y(18) to create a numeric array variable and reserve enough space for all of its elements.

Finally, there are number-storage and word-storage places created and used by Apple PILOT itself, but accessible to you, too. These places are called "system variables". There are four of these: %A (the Answer Count), %B (stores the last response), %X and %Y (the screen coordinates received by the last Accept Point instruction). You don't usually store things in the system variables yourself, but you can use what Apple PILOT has stored there.

RETURN

Every Apple PILOT instruction must end with a RETURN character. Normall this is not a consideration, since you must press the RETURN key to start the next instruction line. However, the last instruction in a lesson must also be followed by RETURN, or it will NOT BE EXECUTED. This last RETURN is easy to forget, when you are typing a lesson in the Lesson Text Editor.

GENERAL INFORMATION

UPPER AND LOWER CASE LETTERS

Apple PILOT does not care whether you type the letters in a lesson in upper-case (capitals), lower-case, or any mixture, with the following exceptions:

> The Type instruction's object text must be typed exactly as you want it to be displayed on the student's screen.
>
> The Match instruction's object text must be typed exactly as you want it to be compared to the student's Accepted response. See the PRoblem and Accept instructions for details about converting all response letters to upper-case or all to lower-case, for easier Matching.

Remember, the Lesson Text Editor shows true upper-case letters in "inverse video", as black letters on a white square. CTRL-Z shifts only the next letter to upper-case, CTRL-A is the locking shift key for letters.

Since the actual instructions of the lesson are seen only in the Lesson Text Editor, you may wish to type everything in lower-case except an occasional capital letter in a Match or Type instruction's object text. The lower-case letters you type will appear as white capital letters on a black background. On many monitors, this will be easier to read than inverse video.

INSTRUCTION LENGTH

The Lesson Text Editor automatically breaks up any instruction that exceeds 39 characters as you type it in, and continues it (beginning with the continuation-colon) on the next line. These breaks and continuation-colons are ignored by PILOT when executing the instruction, and the instruction is treated as if it had been written on one long line.

In the Lesson Text Editor, nothing stops you from continuing an instruction for dozens of lines. When an instruction is executed, however, only the first 25Ø characters of an instruction are used. Any part of an instruction beyond the 25Øth character is simply ignored, as if it did not exist.

You may safely continue Remark instructions as long as you wish, as they are not executed. All other instructions are limited to 25Ø characters total, not counting the continuation-colons themselves, which are ignored during execution.

BLANK LINES

Blank lines, containing no instruction or continuation-colon, are acceptable between different instructions. These can help make a lesson more readable, when you look at it in the Lesson Text Editor. However, do not put blank lines between one portion of an instruction and a colon-continuation of the same instruction.

RUNNING A PROGRAMMED LESSON

Once you have written an Apple PILOT lesson, you will want to test it, and you will want your students to use it. There are two ways to run a programmed lesson: from Author Mode and from Lesson Mode.

TESTING IN AUTHOR MODE

When you are testing and working on a new lesson, you will want to run the lesson from Author Mode. To do this, you select the Run option from the menu of the Lesson Text Editor, and then type the name of the lesson you wish to run. This option runs the specified lesson on the Lesson diskette in drive 2. When the lesson ends, the system re-starts itself, and returns you to the Main Menu of Author Mode.

For testing, you may wish to insert a few lines at the end of the lesson that allow you to repeat the lesson over again. For example:

```
T: Type "again" to repeat lesson.    (Display this text.)
A:                                   (Accept your response.)
M: AGAIN|again                       (Did you type AGAIN or again ?)
JY: START                            (If Yes, Jump back to the label
                                       START , near top of lesson.)
E:                                   (No, so end the lesson.)
```

You may also wish to use the PRoblem instruction to select one or both of the execution-time command options, GOTO and Escape. These commands let you jump to known portions of your program during execution. See the section Execution-Time Commands, later in this chapter.

When you run a lesson from Author Mode, error messages are displayed on the screen along with the offending line from your lesson. This helps you find problems such as mis-typed instructions, or numbers that are too small or too large. Later, when the student runs the lesson in Lesson Mode, the messages are not put on the screen. See the appendix Error Messages, for an explanation of the various error messages.

RUNNING IN LESSON MODE

When your student uses the lesson, the system is started up in Lesson Mode, with the Lesson diskette in drive 1. If you have updated the diskette's HELLO program to list the new lesson, the student can select the proper lesson from the HELLO menu. Otherwise, you must tell the student which lesson name to type.

If you do not want the student to choose from a list of possible lessons, you can name your new lesson HELLO, and that lesson will be run automatically each time the system is started up in Lesson Mode. Or you can write a simple, one-line HELLO lesson consisting of an instruction such as this:

```
LX: BUTTERFLY
```

This HELLO lesson will automatically run the lesson named BUTTERFLY (or whatever name you supply) each time the system is started up in Lesson Mode. See the Link instruction, in the chapter Control Instructions, for details about how this instruction works.

Before giving the new lesson to your students, you may wish to remove any special testing aids, such as the lines shown above which let you repeat the lesson. You may also wish to change any PRoblem instructions that led you to use the execution-time commands GOTO and Escape during testing.

The error messages displayed on the screen when running a lesson from Author Mode are not displayed in Lesson Mode. Any problem which would cause an error message and raise the Error Flag in Author Mode raises the Error Flag without any message in Lesson Mode. Your instructions can use the Error conditioner to test the state of the Error Flag at any point. This lets your lesson's error-handling instructions take care of correcting problems which arise in the course of a normal "conversation" with the lesson.

EXECUTION-TIME COMMANDS

In addition to the Apple PILOT instructions that you program into a lesson when you are in the Lesson Text Editor, there are two special commands that you or your student can use when the program is running:

GOTO

The GOTO command is enabled by the C control option in a PRoblem instruction. After PR:C , any Accepted response that begins with the letters GOTO causes a jump to the destination following the GOTO . The destination may be a label, @M (next Match instruction), or @P (next PRoblem instruction).

@ (ESCAPE)

The Escape command is enabled by the E control option in a PRoblem instruction. After PR:E , any Accepted response that begins with the symbol @ causes a jump to the subroutine labelled SYSX . This is equivalent to the Use instruction U:SYSX . You must have written a subroutine labelled SYSX into your lesson.

For details about selecting these command options, see the description of the PRoblem instruction, in the chapter Response Instructions. For information about using the commands, see the chapter Execution-Time Commands.

CTRL-C

If a CTRL-C is typed in the response to any Accept instruction, the lesson is terminated and the system re-starts itself. To type CTRL-C, type C while holding down the CTRL key. This is a handy way to end any lesson.

RESET

Any time the RESET key on the Apple is pressed, the system re-starts itself. You should avoid pressing the RESET key when the disk drive's red "In Use" light is on, but any other time is fine.

CHAPTER 2

TEXT INSTRUCTIONS

32 R: REMARK

34 T: TYPE
34 In the Lesson Text Editor
34 On the Student's Screen
36 The Text Window
37 Variables
39 Modifiers
40 Example

R: REMARK

R: any text

The Remark instruction allows you to place remarks or comments to the author, or to anyone who might read your lesson in the Lesson Text Editor, into the program text of your PILOT lessons. These comments are not executed during the running of the lesson, and are never seen by the student. For example, if you want to leave yourself a note about when you did the last revision of a lesson, you could insert the instruction:

```
R: THIS IS VERSION #3, FINISHED 3/12/81
```

When you are typing a Remark instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation-colon) on the next line. Your remark is broken only at the spaces or hyphens between words, and may be continued indefinitely. You may also break the Remark line at any point, yourself, by pressing the RETURN key. If you then wish to continue the Remark, you must begin the new line with an R: or a continuation-colon.

You can use remarks to remind yourself how a lesson section works, or what labels and variables are used, or what day you wrote this version of the lesson...anything you or another author might want to know later on, when looking at your lesson in the Lesson Text Editor or on paper. A well-Remarked lesson is much easier to understand. This is important if you wish to share your lessons with other authors, but it is also important for you, when you wish to re-read or modify your own lessons, perhaps months after writing them.

Occasional use of blank Remark lines (lines with no text after the colon or after the continuation-colon) will help to increase readability of your lessons by letting you indicate breaks in the material. To call attention to new sections or important remarks, you might set them off with Remark lines including rows of asterisks (****).

Note: you may also use blank lines with no instruction or continuation-colon on them to make a visual break between instructions, but such blank lines must not separate portions of a continued Remark instruction.

EXAMPLE

```
R: ************************
: *    LESSON AUSTRALIA    *
: *  VERSION 3 : 23 FEB 82 *
:  ************************
:
: TELLS ABOUT THE ABORIGINES
: OF AUSTRALIA, AND NATIVE
: ANIMALS.  REQUIRES GRAPHICS
: FILES "KOALA" AND "DUCKBILL",
: AND SOUND FILE "BOOMERANG".
: LINKS TO LESSON "AUST-TEST".


*INTRO
T: Halfway around the world ...
 ..
...
R: SECTION ON THE KOALA BEAR
*KOALA
 ..
R: COMPUTE TEST SCORE, BASED
: ON NUMBER OF GUESSES (%A).
C: S = 11 - %A


T: You get #S  points.
```

(The first 11 lines are just comments to the author. They give information that might be useful to know, but might take a while to find by reading the entire lesson program. These line of text are never seen by the student when the lesson is running.)

(Blank line inserted for easier reading in Editor.)
(Label for this section.)
(First text student sees.)

(Remark identifies section.)

(Remark tells method of scoring a question.)
(Compute score S by subtracting the Answer-Count %A from 11.)
(Display text, substituting the score S for #S and one space .)

T: TYPE

T: any text or spaces, including simple-numeric or string variable-names

When the Type instruction is executed, the text following the instruction's colon is displayed on the student's screen. A one-line Type instruction usually displays its object text just as it appears in the Lesson Text Editor. A colon-continued Type instruction displays its text formatted to fit the screen, breaking the displayed text at the spaces between words. After the text has been displayed, the cursor jumps to the beginning of the next line on the screen, unless you use the H (Hang) modifier described later in this chapter.

IN THE LESSON TEXT EDITOR

When you are typing a Type instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation-colon) on the next line. Your text is broken only at the spaces or hyphens between words, and may be continued indefinitely. You may also break the Type line at any point, yourself, by pressing the RETURN key. If you then wish to continue the same Type instruction, you must begin the new line with a continuation-colon. In the Lesson Text Editor, blank lines (with no instruction or continuation-colon) must not separate continued lines of a Type instruction.

Note: the text which appears on the student's screen may or may not be formatted like the text which appears in the Lesson Text Editor. All text displayed by the Type instruction is automatically re-formatted when it is displayed on the student's screen, according to the rules of display formatting discussed in the next section.

When Apple PILOT executes a Type instruction that occupies more than one line in the Lesson Text Editor, it ignores all breaks and continuation-colons in the instruction, treating the instruction as if it had been typed on one long line. Only the first 250 characters (about six full lines in the Lesson Text Editor) of a Type instruction are used, not counting the continuation-colons. Any part of an instruction beyond the 250th character is simply ignored, as if it did not exist.

If you want more than 250 characters of text to be formatted into one paragraph, break the text into a sequence of Type instructions and use the Hang modifier on each Type instruction until the last one whose text belongs in that paragraph. The Hang modifier is discussed later in this chapter.

ON THE STUDENT'S SCREEN

If the Type instruction occupies a single line in the Lesson Text Editor, and the displayed text fits on one line of the student's screen, the text appears on the screen just as you typed it in the Lesson Text Editor. For example, these instructions:

```
T:          Hi there!
T:
T:How      are      you      today?
```

would cause this display on the student's screen, when the lesson containing the instructions is run:

```
                Hi there!

How      are      you      today?
```

Notice that you can display a blank line on the screen by using a one-line T: instruction with no text after the colon.

If the Type instruction is colon-continued onto additional lines in the Lesson Text Editor, or whenever the text displayed by a single Type instruction exceeds the width of the student's screen, the text is automatically "formatted" and arranged into continuous lines to make a "paragraph" as it is put on the student's screen. The appearance of the text as displayed on the student's screen may be very different from the appearance of the Type instruction in the Lesson Text Editor. For example, this long Type instruction:

```
T When I was but a
: little boy,
:my father said to me, "The
:letter
:
: which comes after  A  is,
:sure, the letter  B ."
```

would cause this display on the student's screen:

```
When I was but a little boy, my father
said to me, "The letter which comes
after  A  is, sure, the letter  B ."
```

What happens on the student's screen is this: the Apple displays each word and space of your Type instruction's text, ignoring the breaks and continuation-colons, until it comes to the first word that won't fit on the current line of the screen. Then, it ends the current line at the last space, and puts the next word at the beginning of the next line.

If a word is so long that there is no space in an entire line on the screen, the Apple simply breaks the word at the end of the line. If you have typed multiple spaces in your Type instruction's text, they will be displayed faithfully unless they happen to fall at the beginning of a line on the screen. When a single Type instruction is displaying its text, every screen line after the first one will start with a non-space text character. For example, this instruction:

```
T:          Oh the cows fly          up,
:             and the flies cow down
:upontheirtinykneesandshuffleofftoBuffa
:lo
```

will cause this display on the student's screen:

```
          Oh the cows fly        up,
and the flies cow
downupontheirtinykneesandshuffleofftoBu
ffalo
```

In this example, you can see that the spaces before the text on the first line of the instruction are displayed faithfully on the screen. This lets you indent the first line of a paragraph, if you wish. It also allows you to place any amount of text in the middle of the screen, for instance, just by sticking to one-line Type instructions.

The spaces between the last two words in the first line were also displayed just as typed, because they happened to fall in the middle of a screen line. The next spaces, at the start of the second line in the Lesson Text Editor, would have fallen at the start of a new line on the screen, so those spaces were not retained in the display. In the text displayed by a single Type instruction, every screen line after the first must begin with a non-space character.

Finally, we forgot to type a space after the word "down" or before "upon" (either place would do, to separate the two words), so "down" became part of the extremely long word which followed. That word was too long to fit on the current (second) screen line, so it was displayed starting on the next line. When it completely filled that line, there were no spaces at which to break, so the word itself was broken at the end of the screen line.

THE TEXT WINDOW

The Type instruction always formats its object text to appear on the screen within the currently set text window. If a line of displayed text exceeds the width of the text window, the text is automatically broken at a space between words and continued at the beginning of the next line. If the next line would be beyond the bottom line of the text window, the text in the window is "scrolled" up to accomodate the new line at the bottom, while the old top line disappears.

The default text window is the full screen, 40 characters wide by 24 lines high. See the discussion of the Graphics instruction's Viewport command for details about changing the text window. This command lets you do graphics on one part of the screen, while restricting text to a window on another part of the screen. Your Type instructions will automatically re-format the displayed text to fit in the new text window, without any work on your part. For example, if a single Type instruction displayed its object text like this in the normal, full-screen text window:

```
You were roaming in the gloaming, making
loans but never owning: who could guess
that you'd be moaning when the clone at
home stopped phoning?
```

the text would be re-formatted for you to look like this, in a narrower text window extending from character position 10 to character position 30 on the screen:

```
You were roaming in
the gloaming, making
loans but never
owning: who could
guess that you'd be
moaning when the
clone at home stopped
phoning?
```

Note that this re-formatting takes place automatically, without changing anything in your Type instruction.

VARIABLES

Any number of string-variable names and simple numeric-variable names may be placed in the object text of a Type instruction. When the instruction is executed, the string or number currently stored in each named variable is substituted where that variable's name was placed. You can use this feature to individualize your lessons by incorporating the student's name, responses, scores, and other personal or changeable data into the displayed messages. See the chapter Hints for Beginners if you are not familiar with the use of variables for such purposes.

However, the Type instruction sometimes must type the variable's actual name, and at other times it must substitute the word or number currently associated with that variable-name, depending on which you want the student to see. To let the Type instruction distinguish between these two cases, a variable-name destined for replacement by value must be placed in the object text according to certain strict rules:

1. Before it can be replaced by its stored value in a Type instruction:
   a) a string variable must be created using a Dimension instruction and then a Compute or Accept instruction must store a name, word, or other string of characters in that variable;
   b) a simple numeric variable must be created by storing a number in it, using a Compute or Accept instruction.

   For example, to make the string variable F$ and the simple numeric variable K ready for use in a Type instruction, you could use these instructions:

```
D: F$(3Ø)                    (Create string variable F$
                              with enough reserved space to
                              store up to 3Ø characters.)
C: F$ = "miles per hour"     (Store string  miles per hour
                              in string variable F$ .)
C: K = 55                    (Create simple numeric variable
                              K  and store the number 55
                              in it.)
```

2. When it appears for replacement by value in a Type instruction:
   a) a string-variable name must be immediately preceded by a dollar sign (for example, the variable S$ would appear as S);
   b) a simple numeric-variable name must be immediately preceded by a pound sign (for example, the variable N would appear as #N).
   c) each variable name must be immediately followed by a space unless the last character of the variable name is the last character in that Type instruction. The required space after the variable name is NOT displayed on the screen. If you want a space to follow the substituted value on the screen, the variable-name should have TWO spaces after it.

   For instance, if we continue the previous example:

```
T: The speed limit is #K  $F$ .   (Display this text, replacing
                                  #K  and first following space
                                  with the number now stored
                                  in simple numeric variable
                                  K , and  $F$  plus the first
                                  following space with string
                                  now in string variable  F$ .)
```

   This instruction, when the complete example is run, will cause this display on the student's screen:

   The speed limit is 55 miles per hour.

3. Substring variables, string pseudo-variables, and numeric array variable elements (all variables written with subscripts) are not allowed to appear for value-replacement in Type instructions. The value of a subscripted variable should first be assigned to a string variable or simple numeric variable which can then appear in the Type instruction.

   For example, to display the number already stored in numeric array variable element G(13) , you could use these instructions:

```
C: T = G(13)                 (Make a copy of the number now
                              stored in element 13 of
                              numeric array variable  G
                              and store that number in
                              simple numeric variable  T .)
T: Your score is #T .        (Display this text, substituting
                              number now stored in simple
                              numeric variable  T  for  #T
                              and first following space.)
```

Any variable names which you place in the Type object text, but which do not strictly follow the three rules above, will be displayed just as you typed them in, like any other text.

MODIFIERS

TH: The Type instruction normally displays its object text, and then moves the cursor to the beginning of the next line on the screen. The cursor is normally invisible, except when an Accept instruction is waiting for the student to type a response, but it determines where the next character will be put on the screen. This means that the next text displayed by a Type instruction, or the next typed response accepted by an Accept instruction, will normally appear starting on that new line of the screen. For example, these instructions:

```
T:What is your name?         (Display this text.)
A:                           (Accept student's response.)
```

Might result in this display, after the student responds:

```
What is your name?
MACGUILLICUDDLE
```

When the Type instruction is modified by the Hang modifier, the cursor remains at the first character position following the displayed text rather than advancing to the next line. Thus, the next text displayed by a Type instruction, or typed in a response by the student, will appear immediately following the text displayed by the Hang-modified Type instruction. Repeating the previous example, this time using the Hang modifier:

```
TH: What is your name?       (Display text, leaving cursor
                              at the end of the text.)
A:                           (Accept student's response.)
```

Might result in this display:

```
What is your name?MACGUILLIUDDLE
```

To separate the question from the answer, just type a space or two after the question mark in the Type instruction. The space will be correctly displayed, along with the rest of the question.

The Hang modifier is useful any time you want the student's reply to a question to appear on the same line with the question. You can also use this modifier to join several Typed portions of a sentence into one, continuous sentence, with different portions used under different conditions. See the example at the end of this section.

A sequence of unmodified T: instructions will display its text as a sequence of short paragraphs, each paragraph limited to a maximum of 248 characters (25Ø, including the T: itself). If you want a sequence of Type instructions to format its text into one long, continuous paragraph, use the Hang modifier on all but the last Type instruction in the sequence. Each TH: instruction is limited to 25Ø characters, of course, but the combined text displayed by the sequence of instructions may be much longer.

TX: When the Type instruction is modified by the eXecute modifier, and a diskette filename appears in the object field, the character set stored in that diskette file is read into the Apple. From then on, Type instructions and student responses use this new character set instead of the usual one. The normal character set is resumed when another TX: instruction is issued, this time with no filename in the object field. The eXecute Character-Set option is more fully discussed in the chapter called Special Effects Instructions.

EXAMPLE

```
D: A1$(2Ø)                    (Reserve space for storing up
                               to 2Ø characters in string
                               variable A1$ .)
D: S(2Ø)                      (Reserve space for numeric
                               array variable S , with 21
                               elements from S(Ø) to S(2Ø) .)
C: A1$ = "Marion"             (Store the string Marion
                               in string variable A1$ .)
T:Now, let's test your        (Display this text, formatted
: ability to add               to fill the screen lines.)
: two numbers together.
T:                            (Display blank line on screen.)
T:Complete the following:     (Display this line of text.)
*PROB1                        (Label for this section.)
TH:    237 + 49 =             (Display this text, leaving the
                               cursor after the equals sign.)
A: ØN                         (Accept the student's response,
                               which appears after the = ,
                               and store the response's
                               first number in simple
                               numeric variable N .)
T(N<>286) No, $A1$ , try again.
                              (If response number stored in
                               N is not equal to 286,
                               display this text: "No,
                               Marion, try again." Note
                               required space after $A1$ .)
JC: PROB1                     (If N is not equal to 286
                               ( C is the last-expression
                               conditioner), Jump to label
                               PROB1 .)
C: S(1) = 11 - %A             (Compute score, based on number
                               of guesses given by the Answer
                               Count stored in the system
                               variable %A , and store it
                               it element 1 of array S .)
C: X = S(1)                   (Assign element S(1)'s value to
                               simple numeric variable X ,
                               which can appear in Type text.)
T:Yes, $A1$ , 286 is right.   ("Yes, Marion, 286 is right.")
TH Your score of ØX is        (If score X = 2 , displays
TH(X<9): not                 "Your score of 2 is not very good."
T: very good.                  If score X = 1Ø , displays
                             "Your score of 1Ø is very good." )
*PROB2                        (Label for next section. )
```

CHAPTER 3

RESPONSE INSTRUCTIONS

44 PR: PROBLEM
44 The Control Options
45 Response-Editing Options
45 Response-Timing Option
46 Execution-Time Command Options
46 Wipe-Labels Option
47 Programming Notes
47 Example

48 A: ACCEPT
49 Response Timing
49 Response Editing
51 Answer Counting
52 Variables
54 Modifiers
55 Programming Notes
55 Example

58 M: MATCH
59 Controller Characters
60 Editing of Match Text and Responses
60 Modifiers
61 Programming Notes
62 Example

PR: PROBLEM

PR: no object, or list of all control options selected

The PRoblem instruction invokes any of seven different control options, which change certain aspects of lesson execution starting at that point. All but one of these options change how the Accept instruction handles student responses. See the Accept instruction, in the next section of this chapter, for more details.

The other main use for PRoblem instructions is to mark the beginning of new sections of the lesson. Branching instructions may then use these PRoblem instructions as unlabelled destinations, whenever you want execution to jump to the beginning of the next section.

If the Problem instruction includes an object field, it must list all of the control options that will be active in the next portion of the lesson. Any control options not specified are disabled; a PRoblem instruction which selects any new option must also re-select any old options that are to remain in effect.

The instruction PR: , with no object field, marks the beginning of a new lesson section without changing any control options selected by a previous PRoblem instruction. Because the object-less PR: instruction has no effect on the control options previously set, you can easily use this instruction anywhere in a lesson as a section marker and unlabelled destination for branching instructions.

THE CONTROL OPTIONS

Each of the seven control options is specified by a single letter (for the Time option, the letter is followed by an integer). Any number of these option specifiers may appear in the PRoblem instruction's object field. They may be typed in any order, with or without spaces between them.

Option	Specifier	Effect on the Lesson
Lower-case	L	Converts all response letters to lower-case.
Upper-case	U	Converts all response letters to upper-case.
Spaces	S	Removes all spaces from responses.
Time	Tn	Sets a maximum time of n seconds for responses, where n is an integer from 1 through 32767 (TØ = default: unlimited time). If the response has not been completed after n seconds, the response typed so far (or current position of the target of the paddle controls) is accepted anyway.
GOTO	G	Any response of GOTO xxx causes a J:xxx . Lets you or the student use the GOTO execution-time command to Jump to other parts of the lesson.
Escape	E	Any response beginning with @ causes a U:SYSX . Lets you or the student use the Escape execution-time command, which Uses your subroutine labelled SYSX .
Wipe-labels	W	"Forgets" all previously established labels, from this point forward. This lets you use the same label names in the next section as you used in the last section, without confusion.

To select the GOTO, Lower-case, Spaces, and Time option (for 1Ø seconds maximum response time), you could use an instruction such as

```
PR: G L S T1Ø
```

or

```
PR:LT1ØGS
```

These are just two examples of many possible equivalent instructions.

To change the response time subsequently to 2Ø seconds, without changing the other options, you might use an instruction such as this:

```
PR: SLG T2Ø
```

Response-Editing Options

The Lower-case, Upper-case, and Spaces options can be used to remove some of the the variability between different students' responses, for easier Matching. Using either the L control option (converts all response letters to lower-case) or the U option (converts all response letters to capital letters) lets you successfully Match responses whether they are typed in upper-case, lower-case, or any mixture. If both the L and the U control options are specified in one PRoblem instruction, response letters are converted to Upper-case. The S option lets you Match a response whether the student types it as one word or as two words separated by a space. See the discussions of Accept and Match, in this chapter, for more information on using these options.

Response-Timing Option

You can use the Time control option to encourage rapid answers or to prevent a student from spending excessive time on one problem. This option specifies a maximum number of seconds for completing any Accept instruction. If the student has not pressed the RETURN key (or a game-control button, in an Accept Point instruction) after the specified time, the response as it exists at that moment is Accepted. After such a response time-out, the TIM function (see the chapter Advanced Programming) will return the value Ø to indicate the student did not

complete the response within the set time. The Time option is discussed more fully with the Accept instruction.

The T specifier must be followed by a number from Ø through 32767.9 , or by the name of a simple numeric variable which currently holds a number from Ø through 32767.9 . Only the integer portion of this number is used, discarding any portion to the right of the decimal point. If the specified number is Ø , no time limit is imposed. If more than one Time control option appears in a PRoblem instruction, only the rightmost option is used.

Execution-Time Command Options

The GOTO and Escape control options let you or the student issue certain commands when the lesson is running, in place of the usual response to any Accept instruction. If the GOTO option has been selected, an Accepted response of the form

GOTO destination

or goto destination

causes a Jump to the specified destination (usually a label). If the Escape option has been selected, any Accepted response which starts with the character @ causes a U:SYSX , which Uses an author-supplied lesson section or "subroutine" labelled SYSX . The chapter Control Instructions discusses the Jump instruction, and the use of subroutines with instructions Use and End.

For details about the GOTO and Escape options, with examples of their use, see the chapter Execution-Time Commands.

Wipe-Labels Option

The Wipe-labels option, W , can be useful when you copy a portion of an old lesson directly into your new lesson, using the Lesson Text Editor's Copy-from-File command. If the old lesson portion uses some of the same labels as your new lesson, you can give the duplicate labels new names, or you can start the inserted portion with a PR:W instruction.

You can also use the Wipe-labels option if you have so many labels that your lesson takes a long time to find them (this does not happen until you have at least 5Ø different labels). Just insert a PR:W instruction where your lesson completes a section to which it will never need to return. All the labels used so far will be "forgotten", and a new set of labels (with the same or different names) can be started. However, note that branch instructions can not jump to a label which precedes the PR:W instruction. As far as the program is concerned, those labels don't exist anymore.

The W control option eliminates all labels from the Apple's current reference table of label locations, but normally the Apple will search through your lesson to find any label not in its quick-reference table. A search for a label that is not in the Apple's label table will not proceed backward through the lesson beyond a PR:W instruction which has been executed. For branching operations, it is as if the entire lesson portion preceding the PR:W instruction has disappeared.

However, you can jump to any label in the lesson, on either side of the PR:W instruction, BEFORE the PR:W instruction is actually executed. This might be used, for instance, by putting conditions on the PR:W instruction so that it is not executed until, say, the third time through a certain section of the lesson, making it impossible to return to that section a fourth time.

PROGRAMMING NOTES

A PRoblem instruction may be used as an unlabelled, branch destination for Jump, Use, and End instructions, as well as for the GOTO execution-time command (if enabled). When one of those instructions contains the destination @P , program execution jumps to the next PRoblem instruction. If you start each major section with a PR: instruction, the GOTO @P execution-time command will let you or the student jump to the next section at any Accept instruction, even if you don't know any specific label names. Also, branching instructions in the lesson can easily jump to the next section without referring to a particular label name. This lets you re-arrange lesson sections freely without altering any branching instructions. Note that when the PR: instruction is used as an unlabelled destination you type @P , not @PR as you might guess.

EXAMPLE

```
PR: U                                  (Convert response letters
                                         to upper-case.)
T: What color is the sky?              (Display this text.)
M: BLUE                                (Did student type  BLUE ?)
TY: Yes, on a clear day it is blue.    (If Yes, display this text.)
JY: @P                                 (If Yes, Jump to the next
                                         PRoblem instruction.)
TN: Well, on a clear day it is         (If No, display this text,
: usually blue, isn't it?                and this text.)
...
...                                    (More of this section.)
...
PR:                                    (Mark new section, retaining
                                         conversion of response
...                                      letters to capitals.)
...
```

Without the PR:U instruction, you would not know how to Match the student's responses. The correct response to the first question could be BLUE or blue or Blue or bLue , etc. Following the PR:U instruction, all such responses would be converted to BLUE .

A: ACCEPT

A: no object, or optional simple-numeric and/or string variable-names

The Accept instruction waits until the student types a response and presses the RETURN key (except when used with the PR:Tn instruction or with the Single or Point modifiers described later in this section). The student may type up to 8Ø characters (the typed response is broken if it reaches the right edge of the screen and continued on the next line). It then edits the response according to certain rules and stores the edited response in the system variable named %B , where the response may be compared to possible answers written in subsequent Match instructions, or used for other purposes.

If any typed response includes a CTRL-C (type C while holding down the CTRL key), the lesson is terminated, and the system re-starts itself, returning to the Main Menu (in Author Mode) or to the lesson diskette's HELLO lesson (in Lesson Mode). This provides a handy way to end a lesson at the response to any Accept instruction.

The usual sequence of instructions includes a Type instruction which poses a question or problem, or otherwise elicits a response, followed by an Accept instruction which receives the student's typed response, followed by Match instructions which determine whether or not the student's response contains certain predicted key words. For example:

```
T:How much is  2 + 2 ?          (Display this text.)
A:                              (Accept the student's typed
                                  response, and store it in
                                  system variable %B .)
M: 4|four|FOUR                  (Did student type 4 or four
                                  or FOUR ?)
TY: Very good.                  (If Yes, display this text.)
TN: No,  2 + 2 = 4 .            (If No, display this text.)
```

The response can also be stored at the same time in various simple numeric variables and string variables designated by the author, for later use in the lesson. For example:

```
T:Please type your age.         (Display this text.)
A: #Q                           (Accept student's response,
                                  and store the response's
                                  first number in simple
                                  numeric variable Q .)
```

Using Accept to store responses in variables is discussed later in this section.

RESPONSE TIMING

Normally, the Accept instruction will wait indefinitely for the student to type a response. Execution of the Accept instruction resumes when the RETURN key is pressed. However, if the Tn response-timing option has been selected by a PRoblem instruction, the Accept instruction waits for a maximum of n seconds. The number of seconds n must be an integer from 1 through 32767 , or Ø (unlimited response time) . If the student has not pressed the RETURN key after n seconds, execution of the lesson continues, just as if RETURN had been pressed. In this case, the response Accepted consists of any characters the student had typed before the allotted time ran out, even though the student did not press the RETURN key. For example, if you want the student to respond within 5 seconds, you could use these instructions:

```
PR: T5                          (Set maximum time of 5 seconds
                                  for Accepting responses.)
T: You have 5 seconds to        (Display this text.)
: tell me the sum of  5 + 8  .
A:                              (Accept student's answer,
                                  waiting a maximum of
                                  5 seconds.)
T(TIM(1)=Ø): I'm sorry, your    (If the TIM function returns
: time is up.  The correct        a number equal to Ø, the
: answer is  13 .                 student failed to respond
                                  in 5 seconds, so display
                                  this text.)
```

You can use timed responses to encourage quick answers, or to make progress through a lesson automatic, even if no interactive responses are typed. Use the TIM function to test whether or not the student responded to the last Accept instruction within the set response time. If the last Accept instruction was terminated by exceeding the time set by PR:Tn , and not by the student, the TIM function will return the value Ø . See the discussion of the PRoblem instruction, earlier in this chapter, for details about setting the response-timing control option.

If the student completed the last response within the set maximum time, or if the PR:Tn timing option has not been selected, the TIM function returns the number of seconds the student required to complete the last response. You can use this information in your lessons to score the student on speed of response, to judge the difficulty of certain questions, or to discover which questions might be worded more clearly. The chapter Advanced Programming describes the TIM function.

RESPONSE EDITING

The way a response is edited by the Accept instruction depends, in part, on the control options set up to handle student input. These options are set by the most recently executed PRoblem instruction which contained any control options in its object field. The student's response is usually edited according to the following rules:

1. All spaces before the first non-space character are removed.

2. Any multiple spaces (between characters or after the last character) are compressed into single spaces. If the PRoblem instruction's S option is currently selected, all spaces are removed.

3. If the PRoblem instruction's L option is currently selected, all letters are converted to lower-case (small) letters. If the U option is set, all letters are converted to upper-case (capital) letters. These options do not affect non-alphabetic characters.

In addition, you can specify no response editing at all, by using the eXact modifier, discussed later on. See the section Using Compute for String-Editing, in the chapter Computation Instructions, for further response editing possibilites which can augment those discussed here. The Compute instruction can capitalize the first letter of a stored string, delete or replace all examples of a character, or convert all letters to capitals (like PR:U).

When you try to Match words in a student's response, it is important to know as much as possible about the probable form of that response. Response editing removes some of the random variation from one student's response to another's. For example, converting letters to all capitals or to all small letters (item 3) means you don't have to worry about which way a student will type them.

Without any control options set by a PR: instruction, an un-modified Accept instruction will store this response:

```
My      name is     MAGUILLI
CUDDLE,    so there     !
```

```
My name is MAGUILLICUDDLE, so there !
```

The spaces before "My" have been removed entirely, the multiple spaces between words have been reduced to one space, and all upper-case or lower-case letters have been left as typed.

If you had set the S (Spaces) option, in a PR:S instruction, the same response would have been stored like this:

```
MynameisMAGUILLICUDDLE,sothere!
```

And if you had set both the S option and the U (Upper-case) option, in a PR:S U instruction, the same response would have been stored like this:

```
MYNAMEISMAGUILLICUDDLE,SOTHERE!
```

Finally, if you had used the eXact modifier with the Accept instruction, in a AX: instruction, the same response would have been stored like this, regardless of any control options set by a PR: instruction:

```
My      name is     MAGUILLICUDDLE,     so there     !
```

ANSWER COUNTING

An internal Answer Counter records how many times in a row a student has responded to the same question. This Answer Count is stored in the system variable named %A .

When an Accept instruction is executed the first time, the Answer Count is set to 1. Until a new Accept instruction is encountered, the Answer Count is increased by 1 each time the old Accept instruction is executed again. The Answer Count continues to grow until a different Accept instruction is executed, setting the Answer Count back to 1.

The Answer Count value stored in %A can be used in other instructions for counting the number of times the student has attempted to answer the same question. Counting attempted answers can be useful in scoring the student's effort and in evaluating the clarity of your lesson's questions.

The primary use of the Answer Count is to let the student go on to the next question after a specified number of unsuccessful attempts at answering the current question or problem. This is important, as otherwise the student could remain stuck indefinitely at one question, perhaps one that is ambiguously worded. Instructions whose execution is conditional on the Answer Count value can let your lesson respond differently to each attempted answer. For Answer Counts of 1 through 9, your instructions can use the Answer Count conditioner to test the value of %A automatically. For example:

```
PR: U                          (Convert all response letters
                                 to capitals.)
*START                         (Label for this section.)
T: What is the best feeling in (Display this text.)
: the world?
A:                             (Accept student's response.)
M: LOV                         (Did student type  LOV ?)
TY: Yes, I think so too, even  (If Yes, display this text.)
: though I'm only a computer.
JY: NEXT                       (If Yes, Jump to label  NEXT .)
T1: Hint: it makes you warm and (If wrong answer #1, display
: cuddly.                        this text.)
T2: Hint: it makes you feel    (If wrong answer #2, display
: wanted and secure.             this text.)
T3: Well, to me, the best feeling (If wrong answer #3, display
: is to love and be loved.       this text.)
J3: NEXT                       (If wrong answer #3, Jump to
                                 the section labelled  NEXT .)
```

```
T: Think about it, and then try   (After wrong answer #1 and #2,
: again to guess what I'm           display this text.)
: thinking of.
J: START                          (Jump back to label START .)
*NEXT                             (Label for next section.)
```

See the chapter Overview of the Language for a discussion of the Answer Count conditioner. See the discussion of System Variables, in the chapter Advanced Programming, for information on using %A in other instructions.

VARIABLES

After a response has been edited according to the response-editing options described in a previous section, it is automatically stored in the system variable %B , also called the Answer Buffer. The same string is then assigned to any string variables whose names appear correctly in the object field of the Accept instruction. Finally, the first identifiable number found in the response is assigned to any simple numeric variables whose names appear correctly in the object field.

Any number and any combination of string variables and simple numeric variables can be assigned values by an Accept instruction. The names of these variables can appear in any order, but they must be placed in the object field of the Accept instruction according to certain strict rules:

1. Before its name can appear in the object field of an Accept instruction:
   a) a string variable must be created, with space for storing strings of the expected size, by using it in a Dimension instruction.
   b) a simple numeric variable can be created by the Accept instruction or by using a Compute instruction to store any initial value in it.

   For example, to create the string variable S$ and the simple numeric variable N , in preparation for using them in an Accept instruction, you could use these instructions:

```
D: S$(30)                 (Create string variable S$ ,
                            with enough reserved space to
                            store up to 30 characters.)
C: N = 0                  (Create simple numeric variable
                            N and store initial number
                            0 in it.)
```

2. When it appears in the object field of an Accept instruction:
   a) a string-variable name must be immediately preceded by a dollar sign (for example, the variable S$ would appear as S);
   b) a simple numeric-variable name must be immediately preceded by a pound sign (for example, the variable N would appear as #N).
   c) multiple variable names may appear in any order, and they must be separated by spaces.

For example, when an Accept instruction uses the two variables created in the previous part, it might look like this:

```
A: #N $S$ #Q2                 (Accept the student's response.
                                Store the entire accepted,
                                edited response in string
                                variable S$ ; store the
                                response's first number in
                                simple numeric variable N ;
                                create simple numeric variable
                                Q2 and store the response's
                                first number in it, too.)
```

3. Substring variables, string pseudo-variables, and numeric array variable elements (all variables written with subscripts) are not allowed to appear in the object field of an Accept instruction. The response should first be stored in a string variable or simple numeric variable whose contents can then be stored in a subscripted variable in a Compute instruction. For example, to store a numeric response in element 13 of a previously dimensioned numeric array variable J , you could use these instructions:

```
A: #N                     (Accept the student's response,
                            and store its first number in
                            simple numeric variable N .)
C: J(13) = N              (Make a copy of the number now
                            stored in N and store it in
                            element 13 of numeric array
                            variable J .)
```

If the object field of an Accept instruction contains any variable-name which does not conform to these rules, an error message is given (if you are in Author Mode), the Error Flag is raised, and any remaining variables are not given new values.

If there has been no response, or the student pressed only the RETURN key, %B and any object string variables contain the null string. If no identifiable number is found in the response, any simple numeric variables named in the object retain the value they had before the Accept instruction. If no number is found in the response and there is a numeric variable in the object field, the internal Error Flag is raised. Other instructions can test the state of the Error Flag automatically, by using the Error conditioner (see the chapter Overview of the Language).

To find a number in the Answer Buffer, the characters stored in %B are scanned, starting with the first character and looking for a numeric digit (0 through 9). If a numeric digit is found, the preceding character is checked to see if it is a decimal point or a minus sign. If it is a decimal point, the character before that is checked to see if it is a minus sign. Finally, sixteen characters beginning with the first numeric character (minus sign, decimal point, or digit) are scanned until the first one is found that could not be part of this number. The numeric characters collected to that point are then

converted to a number, and that number is stored in all simple numeric variables whose names appear in the Accept instruction's object field.

If more than sixteen numeric characters occur in a row, any characters beyond the sixteenth numeric character are simply ignored. A capital or lower-case E is included only if the next character is a numeric digit or a + or - followed by a numeric digit. Only the first two digits after an E are used, any further digits being ignored. A number with many digits is rounded to six digits, and may be expressed in "scientific" E-type notation.

Here are some sample Accepted responses, and the number that would be stored in a simple numeric variable appearing in the object field of the Accept instruction:

Response Accepted by A: #N	Number Stored in Simple Numeric Variable N	
4 score and 7 years ago,	4	
Apple is No.1 , 4 me.	.1	
I have 5,864,327 dollars	5	
Re-tread number 13	13	
Mr. Jones is 44	44	
five point seven	Unchanged	(no number)
-12345678901234567890123E-23	-1.23457E14	(1st 16 numeric characters used)
12e-1234567	1.2E-11	(1st 2 digits after E are used)
4E38	Unchanged	(number too large)

If the number exceeds the limits for real numbers, an error message is given (in Author Mode) and the Error Flag is raised (in either mode), and all object numeric variables keep the same number that was stored in them before the Accept instruction. The complete response is stored in any string variables which appear before the first numeric variable in the object. Any string variables which appearafter the first numeric variable are given the null or empty string. Numbers must be in the approximate ranges -3.4E38 to -1.2E-38 , 0 , and 1.2E-38 to 3.4E38 .

MODIFIERS

AX: The Accept instruction may be modified by the eXact modifier to accept the student's response exactly as it was typed, with no editing. AX: suppresses the normal removal of leading spaces, the compression of multiple spaces, and temporarily over-rides the effect of any S, L, or U option set by a PRoblem instruction. The exact, un-edited response is assigned to system variable %B and to any object string variables (see the previous section). All other control options, such as the Time option and the execution-time command options, remain in effect.

AS: The Accept instruction can be modified by the Single modifier to accept a single keystroke as a suitable response. AS: accepts the student's first keystroke and places the exact character typed in the Answer Buffer %B and in any object variables. No response editing is done on the accepted character; any S, L, or U option set by a PRoblem instruction is ignored.

The student does not have to press the RETURN key after typing the single response character. The single character response is accepted as soon as it is typed, and lesson execution resumes. If the RETURN key is pressed as the only response, a space character is accepted. The Escape execution-time command, if enabled, can be used with the AS: instruction.

The AS: instruction does not generate a new line; the cursor remains where the single keystroke left it. The next Type instruction's text, for instance, will begin right after the Accepted character on the screen.

AP: The Accept instruction can be modified by the Point modifier to accept the X and Y coordinates of a graphics screen point selected by the game-control paddles, at the moment when either of the game-control buttons is pushed. AP: stores the coordinates in system variables %X and %Y for use by other instructions. This modification of the Accept instruction is discussed more fully in the chapter Special Effects Instructions.

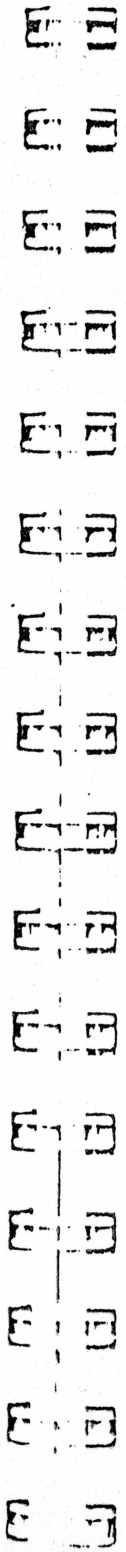
PROGRAMMING NOTES

An Accept instruction may be used as an unlabelled, relative branch destination for Jump, Use, and End instructions. When one of those instructions contains the destination @A , program execution branches back to the last Accept instruction previously executed.

EXAMPLE

```
D: N$(50)                    (Reserve space for string
                               variable N$ to store
                               up to 50 characters .)
T: Please type your name, and (Display this text.)
: then press the RETURN key.
A: $N$                       (Accept response and store
                               it in string variable N$ .)
T: Now, $N$ , type your age. (Display text, substituting
                               the name stored in string
                               variable N$ for $N$ and
                               first following space.)
```

```
A: #A                          (Accept response; create simple
                                 numeric variable A and
                                 store the response's first
                                 number in it.)
JE: ERROR                      (If the Error Flag is up, no
                                 number was typed, so Jump
                                 to section labelled ERROR .)
C: K = 3 * A                   (Multiply age stored in A
                                 by 3, and store result in
                                 simple numeric variable K .
                                 This is the right answer.)
*PROB                          (Label for this section.)
T: What is 3 times your age?   (Display this text.)
A: #B                          (Accept response; create simple
                                 numeric variable B if this
                                 is its first use; store the
                                 response number in B .)
JE: ERROR                      (If the Error Flag is up, no
                                 number was typed, so Jump
                                 to section labelled ERROR .)
T1(B=K): Right the first try!  (If Answer Count is 1 and if
                                 B=K, display this text.)
TC(%A>1): Yes, that's right.   (If B=K (C is last-expression
                                 conditioner) and Answer
                                 Count %A is greater than
                                 1, display this text.)
J(B=K): NEXT                   (If B=K, Jump to label NEXT.)
T(%A<5): Nope, try again.      (Wrong answer: if Answer Count
                                 is still less than 5, display
                                 this text.)
JC: PROB                       (If Answer Count %A is less
                                 than 5 (C is last-expression
                                 conditioner), Jump back to
                                 label PROB for another try.)
T: No, 3 times #A is #K .      (5th bad answer: display text,
                                 substituting age A for #A
                                 and first following space, and
                                 right answer K for #K and
                                 first following space.)
J: NEXT                        (Student has had 5 wrong answers
                                 so Jump ahead to next problem.)
```



```
*ERROR                         (Label for error section.)
T1: Please type a number.      (If Answer Count is 1, display
                                 this text.)
T2: I mean a number like 1, or 23.  (If Answer Count is 2, display
                                 this text.)
T(%A>2): You must type a number.  (If Answer Count %A is greater
                                 than 2, display this text.)
J(1): @A                       (Lower the Error Flag by eval-
                                 uating expression (1), and
                                 Jump back to last Accept
                                 instruction executed.)
*NEXT                          (Label for next section.)
```

In the Accept instruction A: #A , where the student is typing an age, if the student were to respond

MY AGE IS 34 YEARS AND 7 MONTHS.

the Accept instruction would store the number 34 in numeric variable A. However, if the student responds

IN 2 DAYS I WILL BE 34 YEARS OLD

the number 2 will be stored in A .

M: MATCH

M: any text or controller characters

The Match instruction looks for any copy of its object text in the most recently Accepted student response. Subsequent instructions can then use the success of this search, or its lack of success, as a condition for their execution, by means of the Yes and No conditioners. For example:

```
PR: U                                  (Convert all response letters
                                         to capitals.)
T:What color is an orange?             (Display this text.)
A:                                     (Accept student's response.)
M: ORANGE                              (Did student type  ORANGE  ?)
TY:That's right, a ripe orange         (If Yes, display this text.)
: is usually orange-colored.
TN:Well, I mean a nice, ripe orange.   (If No, display this text.)
```

You will generally use the Match instruction to compare a student's response with a response or set of responses that you have supplied. This requires you to predict fairly accurately your students' probable responses, both the correct ones and (often more important) the incorrect ones. Much of your skill and ingenuity as a programmer and teacher will go into this work.

The Match instruction may be used with two modifiers and four special controller characters to produce almost unlimited sets of Match conditions. Simple match conditions are easier to write than complex ones; but simple conditions can allow misinterpretation of responses and thereby lead to confusion.

When you are typing a Match instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation-colon) on the next line. You may also break the instruction where ever you wish, by pressing the RETURN key. If you then wish to continue the Match instruction, you must begin the new line with a continuation-colon. Be sure you do not introduce any unwanted spaces when continuing a Match instruction.

Even though the Editor may break your instruction in the middle of a word, the instruction will be executed correctly. When the instruction is executed, any breaks and continuation-colons are ignored. The object text is treated as if it were one long, unbroken string.

Regardless of the number of lines the instruction occupies, only the first 250 characters of the instruction are actually used at execution time. The continuation-colons are not counted in this total. Any portion of the instruction beyond the 250th character is simply ignored, as if it did not exist.

CONTROLLER CHARACTERS

The Match instruction scans the most recently Accepted student response, looking for any sequence of characters that matches the text in the Match instruction's object field. In the simplest case, the Match object text is a single word or number which the instruction tries to match, character for character, with a word or number in the last Accepted response.

To allow more complex Match conditions, any of four special controller characters may be included in the object text. Any number and combination of these special characters may be used, and in conjunction with any of the modifiers. Each affects the Match in a different way:

% A percent sign is the symbol for a "required beginning or end of Match text item". For a successful Match, the student's response must not have a visible character in the position corresponding to the % sign in the Match text item. Without the % sign, the Match text NO will successfully match the responses I DON'T KNOW , NOW? , I WILL NOT! , SNOW JOB , and TINOWLS . The Match text %NO will still match three of those responses (NOW? , I WILL NOT! , and NONSENSE), but the Match text %NO% will only match a response in which NO has either a space or the beginning of the response in which NO has either a space or the beginning of the response before it, and a space or the end of the response after it.

***** An asterisk is the "wild-card" symbol. For a successful Match, the student's response must contain a character corresponding to each asterisk in the Match text item, but any character at all will Match. Any number of * symbols at the end of a Match text item are ignored. The * in the Match text makes it possible to successfully match response words that have small, predictable spelling errors. For example, the Match text B*C*CLE will match the responses BYCICLE , BICYCLE , and CRAB COCLETTE .

! An exclamation point is the "or" symbol. If an ! appears between two Match text items, a successful Match will occur if either item appears anywhere in the student's response. The Match instruction M: APPLE!PEAR!PLUM!ORANGE will successfully match the responses CRABAPPLE , SPEARMINT , PLUM CRAZY , ORANGES AND ALL , and MY FAVORITE FRUITS ARE APPLES AND BANANAS.

& An ampersand is the "and" symbol. If an & appears between two Match text items, a successful Match will occur only if both items appear in the response, and IN THE SAME ORDER as they appeared in the Match instruction. The Match instruction M: WAY&DOWN&RIVER will match the responses WAY DOWN UPON THE SWANEE RIVER, and SWAYING THE LANDOWNER'S DRIVER, but not I WENT DOWN THE RIVER A WAYS .

To search for more than one Match item in a response, when you don't know the order in which the student will type those items, you must use a series of Match instructions. You can easily make the execution of each Match in the series conditional upon the success of the previous Match instructions by using the Y or N conditioners. Match instructions can also use the Jump modifier, discussed later in this section, to jump quickly through a set of Match instructions looking for the one which applies to the response.

EDITING OF MATCH TEXT AND RESPONSES

Before the Match instruction's object text is actually compared to the student's response, all leading spaces are removed from each Match text item. All the remaining Match text, including any spaces between words, is used exactly as you typed it.

Remember that the Accepted response has also been edited (unless the eXact modifier was used with Accept to suppress all input editing). In general, the A: instruction removes all leading spaces from the response, and compresses all multiple spaces into single spaces. Any input control options selected by a PRoblem instruction may cause further response editing. In particular, be aware of the U option (converts all letters in the response to capitals), the L option (converts all letters to lower case), and the S option (removes all spaces from the response). Note that the AS: instruction also suppresses all input editing on the single character response accepted.

The way you type your Match instruction object text must correspond to the way responses are edited, according to the various defaults and PRoblem instruction options. See the discussions of the Accept instruction and the PRoblem instruction, in this chapter. The effect of these options on your Match instructions is discussed further in a few paragraphs, in the section Programming Notes.

MODIFIERS

Any of the special controller characters may be used in conjunction with any of these modifiers in the same Match instruction. Also, any sensible combination of modifiers may be used on any instruction.

MS: The Match instruction can be modified by the Spelling modifier to allow a successful Match when up to one character of the student's response differs from the corresponding character in the Match item. If multiple Match items are specified, each corresponding response is allowed one wrong character. This modification lets you successfully match the student's response, even though the response contains a minor misspelling. For example, the instruction MS: FIBBER will match the responses THREE FIBBERS , I LIKE FILBERTS , A GIBBERING IDIOT , I9FIB4ER12 , and HIFI BERRIES . You can augment this modifier's action, which also allows one character to be wrong anywhere in the response word, by using the * controller character, which also allows a wrong character at a specific place in the response word.

MJ: The Match instruction can be modified by the Jump modifier to branch automatically to the next Match instruction if the present Jump-modified Match instruction FAILS. The instruction MJ: is equivalent to the instruction M: followed by the instruction JN:@M . Using the MJ: instruction can simplify a series of consecutive Match instructions dealing with a single response; execution will simply "drop through" the unsuccessful Match instructions until it stops at the one which succeeds in Matching the response. For example:

```
PR: U                                  (Convert all response letters
                                         to capitals.)
T: Tell me your favorite color, and    (Display this text.)
: I'll tell you what I think of.
A:                                     (Accept student's response.)
MJ: RED                                (Did student type RED ? If
                                         No, Jump to next Match.)
T: Roses, cherry pies, sunburn.        (Yes, so display this text.)
J: PART2                               (Jump to section PART2.)
MJ: BLUE                               (Did student type BLUE ? If
                                         No, Jump to next Match.)
T: Blueberry pie, distant mountains.   (Yes, so display this text.)
J: PART2                               (Jump to section PART2.)
MJ: GREEN                              (Did student type GREEN ? If
                                         No, Jump to next Match.)
T: Green fields, spinach, waves.       (Yes, so display this text.)
J: PART2                               (Jump to section PART2.)
...                                    ...
...                                    and so on, through the colors
...                                    ...
M: VIOLET                              (Did student type VIOLET ?)
TY: Twilight, deep water, bruises.     (If Yes, display this text.)
TN: Sorry, I don't know that color.    (If No, none of the Matches
                                         was successful, so display
                                         this text.)
*PART2                                 (Label for section PART2.)
```

The example at the end of this section demonstrates the use of the MJ: instruction in more detail.

PROGRAMMING NOTES

A Match instruction may be used as an unlabelled, relative branch destination for Jump, Use, and End instructions, as well as by the GOTO execution-time command (if enabled). When one of those instructions contains the destination @M , program execution jumps to the next Match instruction.

PR: A letter in the Match instruction's object text will not successfully Match a corresponding letter in the student's Accepted response unless both letters are upper-case (capitals) or both are lower-case. Unless you know that all student responses will be typed entirely in upper-case letters or entirely in lower-case letters, you will often wish to convert all response letters to one case or the other. To do this, use the PRoblem instruction with the U control

option (which converts all response letters to upper-case) or with the L option (converts all response letters to lower-case).

EXAMPLE

```
PR: U                                   (Convert all response letters
                                          to capitals.)
T: How many planets are there           (Display this text.)
: in our solar system?
A:                                      (Accept student's response.)
M: %9%!%NINE%                           (Did student type 9 or NINE ?)
TN: No, try again.                      (If No, display this text.)
JN: @A                                  (If No, go back to last Accept.)
T: Good.  Now, try to name the          (Right answer: display these
: nine planets in order, starting         three lines of text.)
: with the one closest to the sun.
A:                                      (Accept student's response.)
MS: MERCURY&VENUS&EARTH&MARS&JUPI       (Did student type all 9 words
:TER&SATURN&URANUS&NEPTUNE&PLUTO           in this order, allowing for a
                                           wrong letter in each word?)
TY: Very good!                          (If Yes, display this text.)
JY: VISIT                               (If Yes, Jump to label  VISIT .)
T(%A<3): No, try it again.              (Wrong answer: if Answer Count
                                           is less than 3, display text,
                                           and Jump back to last Accept.)
JC: @A                                  (3rd wrong answer: display
T: No, the planets are Mercury,            this text, and go on.)
: Venus, Earth, Mars, Jupiter
: Saturn, Uranus, Neptune, and Pluto.
*VISIT                                  (Label for this section.)
T: Which one of the planets             (Display this text.)
: would you like to visit?
A:                                      (Accept student's response.)
MJ: NO                                  (Did student type  NO ?  If
                                           not, Jump to next Match.)
T: Come on, it's vacation time.         (Yes, so display this text,
J: VISIT                                   and Jump back to  VISIT .)
MJ: M*RC*RY                             (Did student type MERCURY ,
                                           allowing misspelling of
                                           letters 2 and 5?  If not,
                                           Jump to the next Match.)
T: Well, Mercury is a warm place        (Yes, so display these lines
: to visit, if you are on the              of text...
: side facing the sun.  You see...
...
J: NEXT                                    ...and Jump to label NEXT .)
MJ: V*N*S                               (Did student type VENUS ,
                                           allowing misspelling of
                                           letters 2 and 4?  If not,
                                           Jump to next Match.)
T: If you like sunny days, Venus        (Yes, so display
: will be a disappointment.  It is         this text...
: always cloudy on Venus, although...
...
J: NEXT                                    ...and Jump to label NEXT .)
```



```
...
...                                     (And so on, trying to Match
...                                       EARTH, MARS, JUPITER, SATURN,
...                                       URANUS, and NEPTUNE.)
...
M: PL*T*                                (Did student PLUTO ,
                                           allowing misspelling of
                                           letters 3 and 5 ?)
TN: That's not one of the planets.      (If No, none of the Matches
:  Try again.                             was successful, so display
                                           this text.)
JN: VISIT                               (If No, Jump back to  VISIT .)
T: Pluto is a very cold place           (Yes, so display
: to visit, probably the coldest           this text...
: you could go in our solar system.
:  Here's why....
...                                        ...and go on to NEXT .)
*NEXT                                   (Label for next section.)
```

Note that the instruction MJ: NO will match any of the following responses successfully: NO PLANET , NONE OF THEM , NOT MERCURY , I DON'T KNOW, and TO JUPITER, RIGHT NOW! . You could make this Match more selective by using the beginning-or-end-of-word controller character % in the instruction MJ: %NO . The instruction MJ: M*RC*RY will match any of these responses successfully: THE PLANET MERCURY , FIRST PLANET:MARCERY , and I'LL HAVE HAM,RC&RYE .

To find out more about using the Match instruction, see the demonstration programs on the lesson diskette provided with your Apple PILOT system. In particular, the lesson MATCH, when you run it, lets you type any kind of Match instruction and then tells you whether your subsequent responses are successfully matched or not. Experiment with this learning tool to get a "feel" for the capabilities of the Match instruction and its various options.

J: JUMP

J: label or unlabelled destination

The Jump instruction changes the sequence of execution in a lesson by branching to the label or to the unlabelled instruction specified in the object field. For example, to Jump to the label REVIEW , this instruction could be used:

```
J: REVIEW                    (Jumps to label REVIEW .)
```

If the specified label is not found in your lesson, an error message is given (in Author Mode) and the Error Flag is raised. Execution then continues at the next instruction. Note that a label used as destination is typed without any preceding asterisk.

UNLABELLED DESTINATIONS

The Jump instruction can use the Accept, Match, and PRoblem instructions as destinations for a Jump, even though these destinations are not preceded by any explicit label:

```
J:@A                 (Jumps back to the last Accept
                      instruction previously executed.)
J:@M                 (Jumps ahead to the next Match
                      instruction.)
J:@P                 (Jumps ahead to the next PRoblem
                      instruction.)
```

If the specified unlabelled instruction is not found in your lesson by searching in the required direction, the lesson is terminated without an error message, even in Author Mode, and the system re-starts itself.

These Jump instructions could use explicit labels as destinations, instead of @A, @M, and @P. However, labels require space and the system requires time to locate and store the locations of the labels. For greater lesson speed and more efficient use of labels, use the unlabelled destinations when possible.

PROGRAMMING NOTES

Like all Apple PILOT instructions, the Jump instruction can be combined with conditioners and expressions in many ways, giving you a large degree of control over the lesson. For example:

```
JY: @P               (Jumps to next PRoblem if the
                      last Match was successful.)
JN: START            (Jumps to label START if the
                      last Match was not successful.)
J3: REVIEW           (Jumps to label REVIEW if the
                      Answer Count is 3 .)

JE: ERROR            (Jumps to label ERROR if the
                      Error Flag is raised.)
J(N>13): @M          (Jumps to the next Match if the
                      assertion that "the number now
                      stored in N is greater than
                      13" is a True assertion.)
JC: @A               (Jumps to the last Accept if the
                      last execution-modifying
                      expression (in this case, the
                      expression N>13 ) was True
                      or non-zero.)
```

See the chapter Overview of the Language for more details about using conditioners, combinations of conditioners, and expressions. See the chapter Execution-Time Commands for information about the GOTO command, which you can use to cause a Jump while a lesson is running.

EXAMPLE

```
PR: U                (Mark new section and convert
                      response letters to capitals.)
*START               (Label for this section.)
C: N = Ø             (Create simple numeric variable
                      N by storing a first number
                      Ø in it, so Accept can store
                      numbers in it, later.)
T: Guess my number, 1 to 99.
                     (Display this text.)
A: #N                (Accept response; store number
                      in numeric variable N .)
JE: ERROR            (If Error Flag is up, no number
                      was typed, so Jump to the
                      section labelled ERROR .)
J(N<1|N>99): ERROR   (If number typed is less than
                      1 or greater than 99, Jump
                      to the label ERROR .)
M: 57                (Did student type 57 ?)
TY: You got it!      (If Yes, display this text.)
JY: @P               (If Yes, Jump ahead to the
                      next PRoblem instruction.)
T(N>57): No, that's too large.
                     (If N is greater than 57,
                      display this text.)
T(N<57): No, that's too small.
                     (If N is less than 57,
                      display this text.)
T(%A<2Ø): Try again. (If Answer Count %A is less
                      than 2Ø, display this text.)
```

```
JC: START                          (If Answer Count %A is less
                                     than 2Ø, Jump back to START.)
T: Too bad.  My number was 57.     (2Øth guess, so display text,
J: @P                                and Jump to next PRoblem.)

*ERROR                             (Label for error routine.)
T: Please type a number,           (Display this text.)
: from 1 through 99 .
J(1): @A                           (Lower Error Flag by evaluating
                                     the expression (1) , and
                                     Jump back to last Accept.)

PR:                                (Mark next section; continue
                                     to convert response letters
                                     to capitals.)
```

Note: the Match instruction M:57 was included to illustrate use of the Jump instruction with the Yes conditioner. You would not normally test a student's numeric response with a Match instruction. M:57 will successfully Match responses such as 57 , 2.3571 , or 57.98 . A better way to test this response would be to use these lines:

```
T(N=57): You got it!               (If response stored in N is
                                     57, display this text.)
JC: @P                             (If response stored in N is
                                     57, Jump ahead to next PR: )
```


U: USE

U: label or unlabelled destination

The Use instruction, like Jump, changes the sequence of execution in a lesson by branching to the label or to the unlabelled instruction specified in the object field. Unlike Jump, however, the Use instruction "remembers" where the branching operation originated, and can return to that spot later, when an End instruction is encountered.

Ordinarily, your lessons employ the Use instruction to execute a program subsection or "subroutine". A subroutine is any program section which i executed after a Use instruction, up to the corresponding End instruction which terminates that section. Understanding the use of subroutines requires knowledge of two instructions: the Use instruction being discussed here, and the End instruction discussed in the next part of this chapter.

Usually, a subroutine starts with an identifying label, and terminates with an End instruction. For example, a subroutine to calculate scores might look like this:

```
*SCORE                             (Label for subroutine SCORE .)
C: P = 5 - %A                      (Compute points P by subtractin
                                     Answer Count %A from 5 .)
T: You received #P points on       (Display this text, substitutin
: this problem.                      value of variable P for #P
                                     and one space .)
E:                                 (End this subroutine.)
```

Each time you want to invoke this subroutine in your lesson, just inser the Use instruction, with your subroutine's label in the object field. Your lesson would execute the subroutine labelled SCORE , in the example above, each time it encountered this instruction:

```
U: SCORE                           (Jump to subroutine SCORE
                                     and return after next E: .)
```

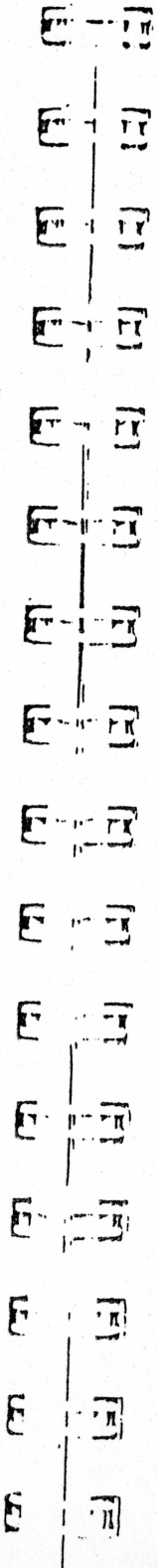
Note that a label used as a destination is typed without any preceding asterisk.

When the U: instruction is executed, program execution branches to the subroutine with the specified label. At the subroutine's End, program execution returns to the instruction following the U: which invoked th subroutine.

If the same labelled program section is encountered without being invoked by a Use instruction, it is not considered a subroutine, and its End instruction will have a very different effect (see the discussion of the End instruction).

```
*TEST
C: Z = X + Y

T: OK, $B$ , what is #X + #Y ?



U: ANSWER

UE: ERROR


T(N=Z): That's right.

EC: @P


T4: No, the answer is #Z .


E4: @P


T: No, try again.
J: TEST

*ANSWER
A: #N $A$ .


M: STOP|HALT|QUIT|END

EY: END

E:


*ERROR
T: You must type a number.
U: ANSWER

JE: ERROR

E:
```

```
(Label for subroutine  TEST .)
(Compute sum of  X  and  Y ,
  and store result in  Z .)
(Display this text, substituting
  student's name for  $B$
  and one space, and
  X and Y values for  #X
  and one space and  #Y
  and one space.)
(Jump to subroutine  ANSWER ,
  and return after next  E: .)
(If the Error Flag is raised,
  Jump to subroutine  ERROR ,
  and return after next  E: .)
(If student's answer N equals
  correct sum Z, display text.)
(If same condition N=Z is true,
  end subroutine by jumping to
  first PR: after  U:TEST which
  invoked this subroutine.)
(Wrong answer: if Answer Count
  is 4, display this text
  substituting Z's value for
  #Z  and one space .)
(If Answer Count is 4, end this
  subroutine by jumping to first
  PR: instruction after  U:TEST
  which invoked this subroutine.)
(Display this text.)
(Jump back to label  TEST .)

(Label for subroutine  ANSWER .)
(Accept student response; store
  complete response in string
  variable  A$  and store the
  response's first number in
  simple numeric variable N .)
(Did student type STOP or HALT
  or QUIT or END ?)
(If Yes, end this subroutine by
  jumping to label  END .)
(End subroutine by jumping to
  instruction after  U:ANSWER
  which invoked this subroutine.)

(Label for subroutine  ERROR .)
(Display this text.)
(Jump to subroutine  ANSWER ,
  and return after next  E .)
(If Error Flag is raised again,
  Jump back to label  ERROR .)
(End subroutine by jumping to
  instruction after  U:ERROR
  which invoked this subroutine.)
```



```
*SCORE
C: P = 5 - %A

C: S3 = S3 + P
T: You received #P  points on
: this problem.  Your total
: score so far is #S3 .

E:


*END
E: END
```

```
(Label for subroutine  SCORE .)
(Compute points P by subtractin
  Answer Count  %A  from 5 .)
(Add P points to total score S3
(Display this text, substitutin
  value of variable P for  #P
  one space and value of variab
  S3 for  #S3  and one space .)
(End subroutine by jumping to
  instruction after  U:SCORE
  which invoked this subroutine

(Label for final section.)
(Because program did not jump
  here from a U: instruction,
  this E: ends the lesson.  The
  label just gives extra
  insurance that the program wi
  end even if some subroutine i
  still active.)
```

Note: if you put an Accept instruction into a subroutine, as this example does in the ANSWER subroutine, there may be a slight delay before the Apple is ready to start receiving the student's response. There are two things you can do to reduce this delay, if it is a problem.

The most straightforward way is to put the Accept instruction immediately after the Type instruction which asks for a response. That way there is no delay while PILOT looks for a labelled subroutine. If you do want to use a subroutine, place it at the beginning of the lesson, rather than at the end. PILOT can locate labels placed in its label table early more quickly than labels placed there later on. However, a label is not placed in the label table until it is actually executed or used.

: END

: no object, or label or unlabelled destination

he End instruction terminates the most recently invoked active ubroutine if any subroutine is being executed, or ends the entire esson if no subroutine is being executed. If no subroutine (reached by he Use instruction) is currently being executed, an End instruction LWAYS ends the lesson, no matter what appears in the End instruction's bject field.

```
E:                              (End the lesson, if there is no
                                  currently active subroutine.)
```

f any subroutine (reached by the Use instruction) is active, an E: nstruction terminates the active subroutine which was most recently nvoked. Usually, a subroutine starts with an identifying label, and erminates with an End instruction. For example, a subroutine to andle an error caused by the student typing a non-number response might ook like this:

```
*ERROR                          (Label for subroutine  ERROR .)
T: You must type a number.      (Display this text.)
A: #N                           (Accept response again, and
                                  assign number to variable N .)
JE:  ERROR                      (If Error Flag is raised again,
                                  Jump back to label  ERROR .)
E:                              (End subroutine by jumping to
                                  instruction after  U:ERROR
                                  which invoked this subroutine.)
```

If the E: has no object, lesson execution branches to the instruction following the U: instruction which invoked the subroutine being terminated. In the same circumstance, an E: with an object causes exactly the same branch to occur, followed immediately by a second branch to the specified label or unlabelled destination.

```
E: PROB3                        (End subroutine by jumping to
                                  the label  PROB3 .)
```

Once a Use instruction invokes a subroutine, that subroutine is considered to be active until a corresponding End instruction is encountered. If subroutines are "nested" (one subroutine invokes another subroutine, etc.), any End instruction terminates the still-active subroutine which was invoked most recently. The outermost subroutine remains active until there has been exactly one End for each Use instruction.

You can nest subroutines up to four levels deep; the fifth U: without an intervening E: causes an error message (in Author Mode) and raises the Error Flag. Lesson execution then resumes at the instruction following the offending Use instruction.

If the label specified by an End instruction is not found, the End instruction is treated as an E: instruction with no object field. An error message is given (in Author Mode) and the Error Flag is raised, and then lesson execution continues at the instruction following the Use instruction which invoked this subroutine.

UNLABELLED DESTINATIONS

The End instruction is usually used with no object, or with a specific label as the object destination. However, it can also use the Accept, Match, and PRoblem instructions as destinations, even though these destinations are not preceded by any explicit label:

```
E:@A                            (Terminate the lesson, or terminate the
                                  most recently invoked subroutine by
                                  jumping back to last  A: instruction
                                  previously executed.)
E:@M                            (Terminate the lesson, or terminate the
                                  most recently invoked subroutine by
                                  jumping to first  M: instruction after
                                  the  U: which invoked that subroutine.)
E:@P                            (Terminate the lesson, or terminate the
                                  most recently invoked subroutine by
                                  jumping to first  PR: instruction after
                                  the  U: which invoked that subroutine.)
```

These End instructions first return to the instruction following the Use instruction which invoked this subroutine, and THEN jump to the specified unlabelled destination, relative to that point in the lesson.

If the unlabelled instruction specified by the End instruction is not found by search through the lesson in the required direction, the lesson is terminated without an error message, even in Author Mode, and the system re-starts itself.

The E: instruction with no object is also, in a sense, using a default unlabelled destination:

```
E:                              (Terminate the lesson, or terminate the
                                  most recently invoked subroutine by
                                  jumping to the next instruction after
                                  the  U: which invoked that subroutine.)
```

These End instructions could use explicit labels as destinations, instead of @A, @M, and @P. However, labels require space and the syste requires time to locate and store the locations of the labels. For greater lesson speed and more efficient use of labels, use the unlabelled destinations when possible.

PROGRAMMING NOTES

When an End instruction is encountered at any point in your program, one of two things may occur:

1. If the number of End's executed (including this one) exceeds the number of Use's executed, the lesson is terminated.

2. If the number Use's executed equals or exceeds the number of End's executed (including this one), the most recently invoked active subroutine is terminated, and lesson execution continues.

In computerese, you can say that Apple PILOT maintains a subroutine return-location "stack". Each time a Use instruction is executed, the lesson location of the instruction following that Use is stored on a stack which can hold up to four of these return locations. Each time an End instruction is encountered, it removes the last return location that was put on the stack, and jumps to that location. Then, if the End instruction has an object destination, it makes another jump to that destination, starting from the point to which it just returned. If an End instruction finds no return location stored on the stack, it ends the lesson.

A subroutine is any program section which is executed after a Use instruction, up to the corresponding End instruction which terminates that invocation. If this same program section is encountered without being invoked by a Use instruction, it is not considered a subroutine, and its End instruction may terminate some other active subroutine or end the entire lesson.

If you place subroutines at the end of the lesson, be sure your lesson is designed to have a definite end point before it gets to the subroutines. Otherwise, program execution may accidentally "fall into" the subroutines, with unexpected results.

If you want to end your lesson, but it is not certain all active subroutines will have been Ended, you might wish to use this ending:

```
*END                          (Label for end section.)
E: END                        (Ends any active subroutine by
                               jumping to label  END , and
                               ends lesson when no active
                               subroutines are left.)
```

EXAMPLE

See the Example for the Use instruction, which precedes the discussion of the End instruction.

L: LINK

L: lesson name, optional label

The Link instruction terminates the current lesson and starts running the lesson specified in the object field. All variables created and used in the lesson containing the Link instruction are maintained and are still available in the new lesson (unless the Erase (X) modifier is used with the Link instruction). Program execution starts at the beginning of the new lesson, if no label follows the lesson name. For example,

```
L: PART-TWO
```

would end the current lesson and start executing the lesson named PART-TWO from the beginning of that lesson. If you wanted the new lesson to start running from the point labelled TEST , the first lesson would use this instruction:

```
L: PART-TWO, TEST
```

The lesson containing the Link instruction, and the lesson specified in the object field of that Link instruction, must be stored on the same Lesson diskette in the normal, one-disk-drive Lesson Mode system. For use in systems having more than one disk drive, see the appendix Using More Disk Drives.

If the specified lesson file is not found on the lesson diskette, an error message is given (in Author Mode) and the system re-starts itself.

MODIFIER

LX: The Erase (X) modifier can be used with the Link instruction to erase the current lesson's variables and release all storage space reserved for those variables before starting the new lesson. If you just break an existing, too-large lesson into two parts, you will probably want to use the L: instruction, without the X modifier, so that the numbers and strings stored in the first lesson portion's variables will still be available to the instructions in the second portion. But if you want to develop and then Link together truly independent lessons, each Dimensioning its own variables and not using information stored by any other lesson, you should use the LX: instruction. If you try to Link independent lessons using the unmodified L: instruction, so much of the available variable space ma be reserved by the first lesson that the second lesson may not be allowed to Dimension or create its variables.

After an LX: instruction, all variables used by the previous lesson are "forgotten", and space reserved for those variables is released for use by the new lesson. If the old lesson was using a special text-character set (see the TX: instruction), the standard ASCII character set is re-loaded for use by the new lesson.

PROGRAMMING NOTES

You can use the Link instruction to join various program portions into a smoothly-running, continuous lesson. It is neccessary to break a lesson into separate lesson sections whenever the lesson becomes too large to be handled in the Lesson Text Editor. The break can come at any point in the original lesson: just end the first section with a Link instruction which starts the next section. Of course, an instruction in either section cannot Jump to a labelled destination in the other section. Since the variables from the first section are still available in the Linked section, you should not re-Dimension any string variables or numeric array variables.

If you create logically distinct lesson sections, these sections can be combined in different ways and joined to other lessons by using the Link instruction. By building up a "library" of useful lesson sections, you can eventually save yourself much programming time. Independent lesson sections, each having its own variables, should be Linked using the LX: instruction.

A large lesson will execute more efficiently if you divide it into separate program segments, and then use L: instructions to jump to major divisions. Often, Jumping to a labelled part of the same lesson is slower than Linking to a new lesson segment, because the Jump instruction must search through all the labels to find its destination. Of course, this advantage is lost if the Link instruction specifies a label in addition to the lesson name.

One use for the LX: instruction is in a HELLO lesson. A HELLO lesson which consists of this one line:

```
LX: DRUMHANGER                    (Start running the lesson
                                    named DRUMHANGER ,
                                    forgetting any variables
                                    now in use.)
```

will automatically run lesson DRUMHANGER whenever a Lesson Mode system is turned on with the diskette containing this HELLO and DRUMHANGER in the primary disk drive.

EXAMPLE

```
*PREPAR                           (Label for this section.)
PR: U                             (Convert response letters to
                                    upper case.)
T: You are about to embark on a   (Display this text.)
: long journey to explore the four
: innermost planets of our solar
: system.  Preparations have been
: underway for several months, and
...
...                               (Getting ready for the trip.)
...
T: Now it is time to tell your ship's
: computer which planet you will visit
: first.  You may choose to visit
: Mercury, Venus, or Mars.
GX: SOLARSYSTM                    (Display stored graphics file
                                    showing a drawing of planets
                                    in the solar system.)

*CHOICE                           (Label for choosing section.)
T: Which planet do you want to    (Display these lines of text.)
: visit first?
A:
MJ: MERCURY                       (Did student type MERCURY ?
                                    If not, Jump to next Match.)
T: OK, let's go visit Mercury.    (Display these lines of text.)
: Better put on your sunglasses,
: because we'll be flying right
: toward the Sun.  Seat-belts
: fastened?  Here we go...
L: MERCURY                        (Terminate this lesson, and
                                    start running lesson named
                                    MERCURY from its beginning.)
MJ: VENUS                         (Did student type VENUS ?
                                    If not, Jump to next Match.)
T: OK, let's go visit Venus.      (Display these lines of text.)
: You may want your rain-coat
: and umbrella, because it rains
: a lot on Venus.  Seat-belts
: fastened?  Here we go...
L: VENUS                          (Terminate this lesson, and
                                    start running lesson named
                                    VENUS from its beginning.)
M: MARS                           (Did student type MARS ?)
TY: OK, let's go visit Mars.      (If Yes, display these lines
: You'll want warm clothes and      of text.)
: something to put over your face,
: because it's cool, and dust
: storms are common.  Seat-belts
: fastened?  Here we go...
LY: MARS                          (If Yes, terminate this lesson,
                                    and start running lesson name
                                    MARS from its beginning.)
T: I'm sorry, that is not one     (No successful Match, so displa
: of the choices.  Try again.       these lines of text, and
J: CHOICE                           Jump back to label *CHOICE .)
```

```
*RETURN                          (Label for section on the
                                  return to Earth.)
```

The lessons named MERCURY, VENUS, and MARS could each end with a similar section for choosing the next planet on the itinerary. After visiting the last planet, students could be returned to the second half of the EARTH lesson by an instruction such as

```
L: EARTH, RETURN
```


XI: EXECUTE INDIRECT

XI: string-variable name

The eXecute Indirect instruction executes the contents of the string variable whose name appears in the object as if those contents had been typed as an instruction in the Lesson Text Editor. An instruction such as a lengthy Graphics or Sound instruction, once you have stored it in a string variable, can be executed easily, anywhere in your lesson, by using the XI: instruction with that variable's name in the object field. You can also use this instruction, in combination with Accept or Compute, to create and execute new instructions during the course of a lesson. For example, to execute the contents of string variable A$ as a PILOT instruction, you could use this eXecute Indirect instruction:

```
XI: A$                           (Execute the contents of
                                   string variable A$
                                   as a PILOT instruction.)
```

Before being used in an XI: instruction, the string variable must appear in a Dimension instruction, and then be assigned a string which is a single, legal PILOT instruction. The instruction stored in the string variable should not begin with a label. The stored instruction may have all the usual modifiers, conditioners, and expressions, in addition to any conditioners and/or expressions which determine the execution of the XI: instruction. You can use either the Compute instruction or the Accept instruction to assign new contents to a string variable.

For example, this program segment:

```
D: A$(42)                        (Reserve space to store up to
                                   42 characters in string
                                   variable A$ .)
C: A$ = "T:What is your name?"   (Store the string
                                    T:What is your name?
                                   in string variable A$ .)
XI: A$                           (Execute contents of variable
                                   A$ as a PILOT instruction.)
```

would cause

```
     What is your name?
```

to be displayed on the student's screen.

Note that the XI: instruction is one of the few instructions which lowers the Error Flag. This occurs just before executing the contents of the string variable.

EXAMPLE

```
T: While this little program is      (Display this text.)
: running, you can type any 4-line
: PILOT program, and execute it as
: many times as you wish.
T:                                   (Display this blank line.)

D: A$(42)                            (Reserve space to store up to
D: B$(42)                              42 characters in each of
D: C$(42)                              these string variables:
D: D$(42)                              A$, B$, C$, and D$ .)

*START                                   (Label for this section.)
T:Type up to 4 instructions, or CTRL-C.  (Display this text.)
T:                                       (Display this blank line.)
T:Press just the RETURN key when         (Display this text.)
: you're done.
AX: $A$                                  (Accept exact response;
                                           store in variable  A$ .)
J(LEN(A$)=Ø): EXEC-3                 (If only RETURN pressed, the
                                       LENgth of the string in  A$
                                       equals Ø, making this the
                                       last instruction, so Jump
                                       to the label  EXEC-3 .)
AX: $B$                              (Accept second instruction.)
J(LEN(B$)=Ø): EXEC-2                 (If last, Jump to  EXEC-2 .)
AX: $C$                              (Accept third instruction.)
J(LEN(C$)=Ø): EXEC-1                 (If last, Jump to  EXEC-1 .)
AX: $D$                              (Accept fourth instruction.)
J: EXEC                              (Jump to label  EXEC .)

*EXEC-3  C: B$ = ""                  (Set B$ to no instruction.)
*EXEC-2  C: C$ = ""                  (Set C$ to no instruction.)
*EXEC-1  C: D$ = ""                  (Set D$ to no instruction.)

*EXEC                                (Label for execution section.)
XI: A$                               (Execute contents of variable
                                       A$  as a PILOT instruction.)
XI: B$                               (Execute  B$  as instruction.)
XI: C$                               (Execute  C$  as instruction.)
XI: D$                               (Execute  D$  as instruction.)

PR:T2                                (Set response time to 2 sec.)
T:Press any key in 2 secs. to stop repeat.   (Display this text.)
AX:                                  (Accept exact response.)
PR: TØ                               (Set unlimited response time.)
J(LEN(%B)=Ø): EXEC                   (If no key was pressed, Jump
                                       to label  EXEC .)
J: START                             (Jump to label  START .)
```

The four instructions are stored in the string variables A$, B$, C$, and D$. If only the RETURN key is pressed, the remaining instructions are skipped. String variables that correspond to no instruction are set to the null string, "". Then the four stored instructions are executed by four XI: instructions. It would be easy to extend this program to accomodate several more instructions.

You can use a program like this to see how various PILOT instructions work, without going back and forth to the Lesson Text Editor between each trial.

Notice how the LEN function is used to test responses: if the LENgth of an Accepted response is zero, no response was typed, or only the RETURN key was pressed. In the timed test-for-repeat section at the end, the AX: instruction was used so that pressing the spacebar can stop the repeat. An unmodified A: instruction would remove a leading space, and LEN(%B) would still be zero.

One of the demonstration lessons provided with Apple PILOT is very similar to this example. The lesson IMMEDIATE lets you execute a single typed PILOT instruction at a time, but you must re-type the instruction in order to repeat it.

W: WAIT

W: number of seconds, or simple numeric variable name

The Wait instruction temporarily stops the lesson for the specified number of seconds, or until a key on the keyboard is pressed, whichever comes first. The length of this wait can be from Ø seconds through 32767 seconds.

You can use the W: instruction to introduce a pause, with a set maximum time, during which the student can read some text on the screen, or examine a graphic display, or solve a problem, or think about a new idea. When the specified time is up, the lesson automatically goes on to the next instruction. Or, if the student wants to proceed before the allotted time has passed, pressing any key will cause the lesson to continue immediately. For example, if you wanted the student to look at a picture on the screen for up to 3Ø seconds, you could use either of these equivalent sets of instructions:

```
T: When you are ready to go          (Display this text.)
: on, press the spacebar.
W: 3Ø                                (Wait up to 3Ø seconds, and
                                       then go on.)
```

or

```
T: Press the spacebar to continue.   (Display this text.)
C: S7 = 3Ø                           (Assign value 3Ø to simple
                                       numeric variable  S7 .)
W: S7                                (Wait up to the number of
                                       seconds specified by the
                                       contents of variable  S7 ,
                                       and then go on.)
```

If you type an actual number in the object field of a Wait instruction, that number should be an integer from Ø through 32767 . If you type a number containing a decimal point or an E , only the part of the number before the decimal or E is used. A number beginning with a decimal point, E , + , or - sign causes an error.

If a simple numeric variable name appears in the Wait instruction's object field, the number stored in that variable can be any real number from Ø through 32767 . Only the integer portion of the stored number (discarding any portion to the right of the decimal point) is actually used. A Wait interval of 32767 seconds or more causes only a very short pause. A Wait interval designated by a negative number causes an error.

The Wait interval can be ended by a key-press which occurs BEFORE the Wait instruction is executed. If more than one key is pressed before a Wait instruction, however, the Apple "forgets" those extra key-presses at the end of the Wait interval. This prevents pre-typed keys from causing a whole series of Wait intervals to be skipped; only the first Wait interval would be ended by keys pressed in advance.

PROGRAMMING NOTES

Do not confuse the Wait instruction with the Time option in the PRoblem instruction. A W:3Ø instruction introduces a pause of up to 3Ø seconds anywhere in the lesson; a PR:T3Ø instruction sets a maximum time of 3Ø seconds for the student to respond to all future Accept instructions. The Wait instruction can be terminated before the set time by pressing any key; a timed Accept instruction can be terminated before the set time only by pressing the RETURN key. Of course, a timed Accept instruction could be used in a similar way to the Wait instruction:

```
T: Press the RETURN key to go on.    (Display this text.)
PR: T3Ø                              (Set maximum response time to
                                       3Ø seconds for Accept.)
A:                                   (Wait up to 3Ø seconds for a
                                       response ended by RETURN.)
PR: TØ                               (Re-set unlimited response
                                       time.)
```

However, the Wait instruction takes fewer instructions and does not leave the cursor on the screen.

EXAMPLE

```
T: Picture yourself on a             (Display this text.)
: tropical island, surrounded
: by bananas and papayas,
: cute little monkeys, and
: beautifully colored parrots.
SX: WAVES                            (Play seashore sounds stored
                                       in file named  WAVES .)
T: Just close your eyes and          (Display this text.)
: imagine you are there...
W: 15                                (Wait up to 15 seconds.)
T: Peering down into the calm        (Display this text.)
: lagoon at your feet, you see
: something that looks like this:
GX: LAGOON                           (Show graphics picture stored
                                       in file named  LAGOON .)
T: Count the number of eyes and      (Display this text.)
: tentacles on this animal.
W: 1Ø                                (Wait up to 1Ø seconds.)
...
...                                  (and so on...)
```

CHAPTER 5

COMPUTATION INSTRUCTIONS

88 D: DIMENSION
88 String Variables
89 Numeric Array Variables
90 Programming Notes
91 Example

93 C: COMPUTE
93 Using Compute for Assignment
94 Constants
94 Variables
96 Expressions
96 Functions
97 Operators
97 Precedence
98 Example
101 Using Compute for String-Editing
103 Example

D: DIMENSION

): name of string variable (maximum number of characters to be stored)
or
): name of numeric array variable (maximum subscript to be used)
or
): name of numeric array variable (maximum subscript1,maximum subscript2)

The Dimension instruction reserves space in Apple's memory for the specified string variable or numeric array variable. Only one variable may be dimensioned in each Dimension instruction. String variables and numeric array variables must first be Dimensioned before they can be used anywhere else in an Apple PILOT lesson. It is not necessary to Dimension simple (non-array) numeric variables.

The Dimension instruction only tells the Apple how much space to save for the variable: it does not store any initial value in the variable. Immediately after Dimensioning, a string variable contains no characters at all, and elements of a numeric array variable contain more-or-less random real numbers. It is your responsibility to store values in the variables you create, by using Compute or Accept instructions.

STRING VARIABLES

For a string variable, designated by a $ following the variable-name, the specified size gives the maximum number of characters which can be stored in that variable. For example, if you want your lesson to store up to 5Ø characters in a string variable named S$, you could use this line at the beginning of the lesson:

```
D: S$(5Ø)          (Reserves space in Apple's
                    memory for storing strings
                    up to 5Ø characters long
                    in string variable S$ .)
```

String variables can be given a maximum size from 1 through 255 characters. Apple PILOT has a total of 6ØØ storage locations for all characters stored by all the string variables in a lesson. The instruction D:R$(1ØØ) reserves 1ØØ character-storage locations for use by the string variable R$. That leaves a maximum of 5ØØ storage locations available for use by other string variables. (In fact, each string variable uses one additional location to save the length of the string currently stored in that variable. Thus, D:R$(1ØØ) reserves 1Ø1 of the 6ØØ possible locations.)

Furthermore, when you store a string in a string variable, that string must first be placed in the UNRESERVED portion of the 6ØØ locations, before it can be stored in the reserved string-variable locations. For example, to store a 75-character string in the string variable R$ dimensioned in the last paragraph, there must be at least 75 character-storage locations not reserved for ANY string variable.

Out of the 6ØØ possible character-storage locations, there should always be at least as much unreserved space remaining as it will take to hold the largest string you will want to store in a string variable. If the largest string variable in a lesson is Dimensioned to a maximum size of 1ØØ characters, make sure the sum of the Dimensioned sizes for all the string variables in your lesson does not exceed 5ØØ characters.

A string variable always reserves the full number of character-storing locations given in the Dimension instruction for that variable. If you store a string of less than the maximum Dimensioned size in the variable, not all of the reserved locations will be used, but the unused locations are still reserved and unavailable for storing characters in any other string variable.

Once a set of character-storing locations has been reserved for a specified string variable by a Dimension instruction, those locations remain permanently reserved for the rest of the lesson. If another Dimension instruction re-Dimensions the same string variable, the first set of locations reserved for that variable remain reserved, but are made completely unavailable, and a new, additional set of locations is reserved.

NUMERIC ARRAY VARIABLES

Numeric array variables are like simple numeric variables, except that a "family" of variables is created, using the same name but followed by a distinguishing number in parentheses. The number that identifies each of the variables, or "elements", in a numeric array is called the "subscript" of that element. For example, the first element of numeric array variable P would be specified as P(Ø) , the second element would be P(1) , the third would be P(2) , and so on. This numeric array variable might be pictured like this:

P	(Ø)	(1)	(2)	(3)	(4)	(5)

Numeric arrays may be one-dimensional (requiring one subscript to specify each element) or two-dimensional (requiring two subscripts to specify each element). If Q is a two-dimensional numeric array variable, its first element would be Q(Ø,Ø) , the second element would be Q(Ø,1) , the third would be Q(Ø,2) , and so on. There would also be more elements such as Q(1,Ø) , Q(1,1) , Q(1,2) , etc., and Q(2,Ø) , Q(2,1) , Q(2,2) , etc. This two-dimensional numeric array variable might be pictured like this:

Q	(Ø,Ø)	(Ø,1)	(Ø,2)	(Ø,3)
	(1,Ø)	(1,1)	(1,2)	(1,3)
	(2,Ø)	(2,1)	(2,2)	(2,3)

When you Dimension a numeric array variable, the specified size gives the maximum array-element subscript allowed for each dimension of the array. The subscript of the first element in a numeric array is always (Ø) or (Ø,Ø) . For example:

```
D: N(3Ø)          (Reserves space in Apple's memory
                    for storing up to 31 real numbers
                    in numeric array variable elements
                    N(Ø) through N(3Ø). This is a
                    one-dimensional array, so only one
                    subscript is needed to specify any
                    of its elements.)

D: V(12,4)        (Reserves space in Apple's memory
                    for storing up to 13*5=65 real
                    numbers in numeric array variable
                    elements N(Ø,Ø) through N(12,4).
                    This is a two-dimensional array, so
                    two subscripts are needed to specify
                    any of its elements.)
```

Each dimension of an array can be given a maximum subscript from Ø through 97. However, there are a total of 98 storage locations for all the numbers stored by all the numeric array variables and all the simple numeric variables in a lesson. This means, for example, that a two-dimensional array whose first dimension has a maximum subscript of 1Ø cannot be given a maximum subscript greater than 7 for the second dimension. Remember that the first identifying subscript in either dimension is Ø, not 1. Dimensioning such a numeric array variable would reserve 11*8=88 number-storage locations out of the possible 98 locations.

A numeric array variable always occupies the full set of number-storing locations specified in the Dimension instruction for that variable. If you store values in fewer than the maximum Dimensioned number of elements of that variable, you will not have used all of the reserved locations, but the unused locations are still reserved and unavailable for storing numbers in any other numeric variable.

Once a set of number-storing locations has been reserved for a specified numeric array variable by a Dimension instruction, those locations remain permanently reserved for the rest of the lesson. If another Dimension instruction re-Dimensions the same numeric array variable, the first set of locations reserved for that variable remain reserved, but are made completely unavailable, and a new, additional set of locations is reserved.

PROGRAMMING NOTES

It is a good practice to place all D: instructions together at the beginning of each lesson so that they are executed one time only. If a Dimension instruction occurs in a program portion which is looped through several times, additional number-storage locations are used up each time the variable is re-Dimensioned, just as if you were Dimensioning new variables. Since there are only 98 locations available for storing numbers, and 6ØØ locations for storing characters, the repeated Dimension instruction might soon cause the message

R-error (Too many arrays or strings)

EXAMPLE

```
D: A$(2Ø)          (Reserve space in memory for
                     storing up to 2Ø characters
                     in string variable A$ .)

D: B(2Ø)           (Reserve space in memory for
                     storing up to 21 real numbers
                     in numeric array variable
                     elements B(Ø) through B(2Ø) .

T: What is your name?   (Display this text.)
A: $A$                  (Accept student's response, and
                         store in string variable A$
C: M = 1                (Store the value 1 in simple
                         numeric variable M . This
                         sets M's initial value.)

T: Okay, $A$ , this will give you   (Display text, substituting
: some practice adding numbers.      student's name for $A$ .)
*TEST                               (Label for this section.)
T: What is #M + #M ?                (Display text, substituting
                                     the number currently stored
                                     in variable M for each #M .

A: #N                               (Accept student's response, an
                                     store the first number in it
                                     in simple numeric variable N
C(N=M+M): B(M) = 1                  (If N=M+M, the student's answe
                                     was right, so store value 1
                                     in numeric array element B(M

C(N<>M+M): B(M) = Ø                 (If N is not equal to M+M, the
                                     answer was wrong, so store
                                     the value Ø in element B(M)
C: M = M+1                          (Increment value of M by 1.
J(M<21): TEST                       (If M is less than 21, Jump
                                     back to the label TEST .)
*SCORE                              (Label for section of lesson
                                     which uses the scores store
                                     in the elements of numeric
                                     array variable B .)
```

Note: more advanced programmers would probably replace the two lines

```
C(N=M+M): B(M) = 1
C(N<>M+M): B(M) = Ø
```

with the single equivalent instruction

```
C: B(M) = (N=M+M)
```

(Evalute the assertion N=M+M . If it is true, store the result 1 in numeric array variable element B(M) . If false, store the result Ø in element B(M) .)

C: COMPUTE

```
C: name of variable = expression
or
C: /name of string variable    editing option    option    option
```

The Compute instruction can be used for two purposes:

1. a) to store a fixed or computed number in a simple numeric variable or in an element of a numeric array variable,
   b) to store a fixed or computed string of characters in a string variable or substring variable,

2. to edit the contents of a string variable.

Examples:

```
C: N = 2 + 3/X
```

(Calculate the current value of 2+3/X , and store it in simple numeric variable N .)

```
C: S$ = "APPLE"
```

(Store the characters APPLE in string variable S$.)

```
C: /S$  /PX
```

(Edit the contents of string variable S$, replacing each character "P" with an "X"; then store the new string back in string variable S$.)

The most common use of the Compute instruction is to do calculations and to store numbers and strings in their appropriate variables (as shown in the first two examples, above). Most of this chapter discusses that kind of use. The final topic in this chapter discusses using the Compute instruction for editing a string (third example, above).

This chapter does not discuss constants, variables, functions, operators, and expressions in great depth. The discussion which follows will let you use some of Compute's features without a detailed understanding of these items. Later, when you are more familiar with Apple PILOT, you may wish to read the chapter Advanced Programming to learn more about the calculating powers available in the Compute instruction and in conditioning expressions.

USING COMPUTE FOR ASSIGNMENT

Only two instructions can store a number or a string in a variable: the Accept instruction and the Compute instruction. Storing a number or a

using a Compute instruction. That way, if there is no number in the student's response you know what default number has been stored in the variable.

You can also create a simple numeric variable by refering to it in the expression on the right side of a Compute instruction's assignment indicator. A simple numeric variable created this way will contain some more-or-less random real number. For example, if this were the first instruction in a lesson:

```
C: X = 5 + R
```

(Add 5 to the number stored in simple numeric variable R , and store the result in simple numeric variable X .)

it would create the simple numeric variable R , add 5 to whatever random number happens to appear in R , create simple numeric variable X , and store the result in X . This kind of situation is most likely to arise when you accidentally refer to a simple numeric variable that you've never used before. Since R might contain any number from 3.4E38 to -3.4E38, the results in your lesson can be quite surprising.

For more information about the different kinds of variables, see the chapter Advanced Programming.

Expressions

An expression may be a constant, a variable, or combinations of constants and variables with items that indicate some calculation or other operation to be done on the constants and variables. There are two kinds of items that indicate an operation, rather than a value. These are "functions" and "operators".

Functions

A function is a short name (usually three letters) which specifies a particular calculation or other operation which is performed on the constant, variable, or expression that appears in parentheses after the function's name. Their use requires some knowledge of mathematics and its use in programs, so we have put their discussion in the chapter Advanced Programming.

Item Name	Item	Examples
Function	Calculated Number or String	SIN(X) , STR(X) , PDL(X)

Example in a Compute instruction:

```
C: A = SIN(1.5)
```

(Calculate the sine of an angle of 1.5 radians, and store the result in simple numeric variable A .)

```
C: X = PDL(Ø)
```

(Store a number from Ø throu
255, indicating the positi
of game control ØØ, in simp
numeric variable X .)

Operators

Operators are used with constants, variables, and functions to indic
all the usual calculator operations.

Operator Symbol	Operation	Example Expressions
+	Addition	99+12.45 , 3.4+PDL(T) , E+
-	Subtraction	16-43.2 , 5-SIN(X) , G-H
*	Multiplication	983*12 , Y*3.14 , B*M
/	Division	2.3/.Ø56 , W6/U , P/SIN(13

Example in a Compute instruction:

```
C: V = D / T
```

(Divide the number stored in
by the number stored in T
and store the result in V

There are also quite a few operators not offered on most calculators These operators let you do exponentiation (multiplying a number by itself a specified number of times), comparisons between two numbers strings, concatenation (joining together) of two strings, and the logic operations AND, OR, and NOT. These operators are all discusse the chapter Advanced Programming.

Precedence

In expressions with several operators, it may not be clear which operation will take place first. For example, the expression 12+4/ might be 12+4 divided by 2 = 16/2 = 8 or it might be 12 plus 4/2 = 12+2 = 14 . The chapter Advanced Programming discusses the special rules for "precedence" among operators, but there is an easier way. You can use parentheses to indicate how you want such an expression be evaluated. The rule is simple: the sub-expressions in the innermost set of parentheses are evaluated first, then those result are used in evaluating the surrounding expressions, and so on.

Example in a Compute instruction:

```
C: D = (17-13) * (5/(2+.5))
```

In this expression, the innermost set of parentheses surrounds the expression 2+.5 , so the first operation is to add .5 to the numbe 2 . That leaves the expression as (17-13)*(5/2.5) . Next, the sub-expressions 17-13 and 5/2.5 must be evaluated (the order do not make any difference), leaving the expression 4*2 . Finally, t result, 8 , is stored in simple numeric variable D . Of course, a the numbers in such an expression could be replaced by a variable n by a function, or by yet another sub-expression in parentheses.

```
                                  labelled  LAST .)
J(X>56Ø): END                     (If x-coordinate  X  is greater
                                    than 56Ø (right edge of the
                                    screen), Jump to the section
                                    labelled  END .)

G: DX,Y                           (Read in the Graphics routines,
                                    and Draw a line to x=X,y=Y .)

C: XØ = X                         (Store old x-coordinate  X  in
                                    simple numeric variable  XØ .)

C: YØ = Y                         (Store old y-coordinate  Y  in
                                    simple numeric variable  YØ .)

J: POINT                          (Jump back to label  POINT , to
                                    calculate a new  X  and  Y .)

*LAST                             (Label for last-point section.)
C: X = (B-YØ)*(X-XØ)/(Y-YØ) + XØ  (Calculate the x-coordinate of
                                    the point where path will
                                    cross the baseline.  Store
                                    that coordinate back in  X .)

C: Y = B                          (Set y-coordinate to height of
                                    baseline B .)

*END                              (Label for end section.)
G: DX,Y;ES;MØ,B                   (Read in the Graphics routines,
                                    Draw a line to x=X,y=Y; Erase
                                    the text window; Move back to
                                    starting position x=Ø,y=B.)

T:Shall I clear the screen before (Display this text.)
: drawing the next path?
T:(Press CTRL-C to stop this lesson.)  (Display this line of text.)
A:                                (Accept student's response.)
M: N!DON                          (Did student type N or DON ?)
JN: CLEAR                         (If No, Jump to label  CLEAR .)
J: SPEED                          (Yes, so Jump to label SPEED .)
```

By using simple numeric variables to store many of the initial conditions, you make it easy to change these conditions as you develop the lesson. For instance, the height of the baseline can be adjusted by simply changing the value stored in the baseline variable B . You make some small sacrifice in speed for this convenience, though, because it takes PILOT extra time to look up the variables' stored values.

The baseline height B should not be made lower than about 86 , or it will be erased by the text lines at the bottom of the screen. To know how high the text goes, in graphic screen coordinates, remember (or see the Graphics instruction, in the chapter Special Effects Instructions) that a character occupies an area 7 screen-dots wide by 8 screen-dots high. Each screen-dot occupies an area specified by about 2 graphic screen x-coordinates wide and about 2 and 2/3 y-coordinates high. There are four lines of text at the bottom of the screen, an area that is four characters, or 4*8=32 screen-dots, or 32*(8/3)=85 graphic y-coordinates high. Thus, any graphic point whose y-coordinate is less than about 86 will probably be in the text viewport.

There are some limits on how quickly this lesson can run. Virtually all of the lesson is spent in executing the nine instructions of the section labelled POINT , over and over again. The Graphics instruction accounts for much of this time, because it must read the Graphics instructions into the Apple from the diskette again every time it is executed. You could, of course, have the lesson calculate the coordinates for several successive points, and then Draw them all in one Graphics instruction. However, the extra calculations would take nearly as much time as re-starting the disk drive, and the student would have nothing to watch during the process.

One thing which will speed up the path-drawing is to change the time-increment variable, I , to a larger value such as 1 . However, this calculates fewer points along the path, so each path is more "jagged". Another small improvement in speed can be made by replacing as many of the variables as you can, in these nine instructions, by actual numbers.

The ending does an "interpolation" to calculate the x-coordinate of the point where the last path segment crosses the baseline. Basically, there is a straight line from the old point x=XØ,y=YØ to the new point x=X,y=Y , and a point x=H,y=B somewhere along it, where the line crosses the baseline. The ratio (B-YØ)/(Y-YØ) is equal to ratio (P-XØ)/(X-XØ) , which means that P = ((B-YØ)*(X-XØ)/(Y-YØ)) + XØ . Apple PILOT does this calculation and stores the value of P as the new number in simple numeric variable X , all in one instruction.

USING COMPUTE FOR STRING-EDITING

In addition to its normal task of evaluating an expression and assigning the result to a variable, the Compute instruction also has a slightly different use: it can do certain quick, easy editing changes to the contents of a specified string variable. The normal Compute instruction follows this form:

```
C: variable name = expression
```

To do string editing, you must use a somewhat different form in typing the Compute instruction's object field:

```
C: /string variable name  option  option  option
```

Note that the string variable's name is preceded by a slash (/), and that there is no "assignment" indicator (=) following the name. Any number of editing options may follow the variable name, and the list of options may be colon-continued onto subsequent instruction lines. For the sake of clarity, it may be helpful to separate the options by spaces.

```
JN: NEXT                          (If No, Jump to label  NEXT .)
T: Sorry.  Try typing the number  (Yes, so display these lines
: of dollars you would like,        of text.)
: again, using numbers only.
A: $B$                            (Accept student's response;
                                    store in string variable B$.)
J: NUMBER                         (Jump back to label NUMBER .)

*NEXT                             (Label for next section.)
```

In this example, we used Compute's string-editing feature twice. When the student types a name, the PR:L instruction at the beginning makes sure it is converted to all small letters. Then the Compute instruction C:/N$ C capitalizes the first letter of the name. Thus, every name comes out with the first letter a capital and the rest small letters, no matter how the name was typed originally.

In processing the student's typed number, we used another Compute instruction to remove all commas. Removing the commas is necessary if you want the FLO function to convert a string such as $3,000,000 to the correct number. With those commas in the string, FLO would return the number 3, not 3E6. Of course, you will still have to explain to the student how to read a response which has been converted to "scientific" E-type notation.

CHAPTER 6

SPECIAL EFFECTS INSTRUCTIONS

106 G: GRAPHICS
106 Graphics Commands
108 Writing Graphics Instructions
110 PILOT Graphics Screen
111 Apple II Hi-Res Graphics Screen
112 Text Screen
113 Text and Graphics Cursors
113 Full Screen (No Viewport)
114 With a Viewport
115 Color
116 Modifier
116 Example

119 GX: EXECUTE GRAPHICS FILE
119 Normal Graphics Files
119 Quick-Draw Graphics Files
120 Example

122 AP: ACCEPT POINT
122 Response Timing
123 Example

126 TX: EXECUTE CHARACTER SET FILE
127 Example

129 S: SOUND
130 Modifier
131 Example

132 SX: EXECUTE SOUND FILE
132 Example

fit on the screen: characters placed beyond the margins of the screen just disappear. To send non-printing characters, use the X command.

Xmit G: Xi,j,k — Sends characters whose decimal ASCII code numbers are i , j , and k to the screen. The decimal ASCII codes are integers from Ø through 127 . Numbers greater than 128 just repeat the same 128 characters over and over.

This command is usually used to send non-printing control characters to the screen. Most of these character-commands affect the text cursor, only. For example, the ASCII code for CTRL-H (back-space) is 8 . G:X8,8,8,8,8 would back-space the text cursor five character-positions, without moving the graphics cursor or causing a display to appear.

Printing characters are displayed starting at the current graphics cursor position, although the T command is usually used for this purpose. The first character sent to the screen following ASCII 27 or 16 is used as part of the screen-control command, and not displayed. The first two characters sent following ASCII 3Ø are used as part of the screen-control command, and not displayed. If an X command does not contain enough character codes to complete the screen command specified by a 27, 16, or 3Ø, the correct number of 32's are supplied. For clarification, see the Appendix ASCII Character Codes.

Viewport G: Vl,r,t,b — Sets left, right, top, and bottom character positions for the text viewport. Affects text displayed by T: instructions only, not text displayed by G:Tabc instructions. The default viewport is the full screen. After you set a smaller viewport, the text cursor and the graphics cursor are maintained independently.

G: V — Equivalent to G:VØ,39,Ø,23 ; re-sets the text viewport to occupy the full screen. The text cursor again follows the graphics cursor after this command.

WRITING GRAPHICS INSTRUCTIONS

The object field of each G: instruction may consist of a single graphics command, or several commands separated by semi-colons (;). Each new G: instruction causes the graphics routines to be read into the Apple again, from the Lesson diskette. The only exception to this is the G: ES instruction. For this reason, it is often advantageous to combine as many graphics commands as possible into each G: instruction.

Spaces within a command are okay, but there must be no space between a semi-colon and the following command letter. Spaces may be substituted for commas.

All numbers may be either integers or simple numeric variables. Coordinates may be separated by a comma or a space.

The values of n (color) may be from Ø through 32767. However, the integers Ø through 9 invoke all the available different colors. Higher numbers are ignored, and do not change the set color.

The values of x and y may be from Ø through 32767. However, the screen display is limited to those absolute x positions (offset x plus relative x) from Ø (left edge) through 559 (right edge) and absolute y positions (offset y plus relative y) from Ø (bottom edge) through 51Ø (top edge).

The values of decimal ASCII numbers may be from Ø through 32767, but those values greater than 127 repeat the set of characters corresponding to the normal ASCII range Ø through 127 . See the appendix ASCII Character Codes for a list of the characters and their associated code numbers and screen commands.

The values of l , r , t , and b (left, right, top, and bottom character positions for the text Viewport) may be from Ø through 32767. However, actual character positions range from Ø (left) through 39 (rightmost), and from Ø (topmost) through 23 (bottom).

Imaginative use of the Graphics instructions G: and GX: can make a dramatic difference in the effectiveness of your lessons. Even a very simple drawing or other visual effect on the screen will give the student new interest in the lesson and relief from reading endless text.

When you are developing a new drawing or other visual effect, it may be easier to use several G: instructions with just one or a few object commands each, rather than one G: instruction with a long list of commands. Changing the sequence of commands is much easier when those commands are not buried in a long command list. However, the separate commands do execute more slowly, because the graphics routines must be read into the Apple again from diskette for each new G: instruction. Later, when you are satisfied with the complete drawing, you can combine the separate instructions into a long list of commands in a single instruction.

If you do type a long Graphics instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation colon) on the next line. You may also break the instruction anywhere you wish, by pressing the RETURN key. If you then wish to continue the

Graphics instruction, you must begin the new line with a continuation colon or a new G: instruction.

Even though the Editor may break your instruction in the middle of a command when it reaches the thirty-ninth character, the instruction will be executed correctly. When the instruction is executed, any breaks and continuation-colons are ignored, whether they were introduced by you or by the Editor. The object text is treated as if it were one long, unbroken string. This means that if you break a graphics instruction, and then colon-continue the list of commands on the next line, you must not leave out any separating semi-colons or introduce any illegal space between a semi-colon and the following command letter.

Regardless of the number of lines a single G: instruction occupies, only the first 25Ø characters of the instruction are actually used at execution time. The continuation-colons are not counted in this total. If the instruction is longer than 25Ø characters, any characters after the 25Øth character are simply ignored.

PILOT GRAPHICS SCREEN

You can think of the student's screen as divided into a grid of vertical columns and horizontal rows. Two different grids are used for the same screen, one by the ordinary Type instruction T: for placing text characters, and the other by the Graphics instruction G: for drawing dots.

The G: instruction can draw points and lines on the screen in positions determined by a grid of 56Ø columns by 511 rows. You can specify any particular screen position by its column and row. The columns are numbered from Ø (the leftmost position) through 559 (the rightmost position). The rows are numbered from Ø (at the bottom of the screen) through 51Ø (the top of the screen).

When you plot a point on the student's screen, your Graphics commands specify that point's location by giving its horizontal position, or x-coordinate, and its vertical position, or y-coordinate, relative to some absolute screen position. The reference screen position is called the offset. By default, the offset position is at column Ø, row Ø. With this offset, the relative x,y coordinates of a point are the same as its actual column,row screen position. Thus this instruction:

```
G: P2Ø,35
```

would plot a point on the screen at column 2Ø, row 35. But you can use the Offset command to change the offset to any screen position. For instance, if you used the Offset command to change the offset to column 1Ø, row 17 , then the same Plot command:

```
G: O1Ø,17
G: P2Ø,35
```

would plot a point on the screen at column 3Ø (1Ø+2Ø), row 52 (17+35). This allows you to draw an image on the screen in one place, and then draw it again in a new place using the same instructions, just by changing the offset position.

The coordinates of a point must always be integers from Ø through 32767 If a simple numeric variable appears in the place of a coordinate, the contents of that variable need not be an integer, but the same limits apply. Since the screen itself is only 56Ø columns wide by 511 columns high, you can draw lines which extend far beyond the actual boundaries of the screen. Such a line just appears at the first actual screen position on its path, and disappears at the last actual screen position on its path.

APPLE II HI-RES GRAPHICS SCREEN

This description of PILOT graphics coordinates is further complicated b the fact that Apple PILOT uses the Apple II high-resolution graphics screen for all displays at execution time. This screen can actually display dots at only 28Ø different horizontal positions, and at only 19 vertical positions.

Normally, you do not have to think about this, as Apple PILOT automatically converts PILOT coordinates into the correct screen-dot locations. However, it is the 28Ø by 192 screen-dot locations which determine the ultimate resolution of your drawings. If you Plot a PILO graphics point at x=4ØØ,y=3ØØ and another at x=4Ø1,y=3ØØ the two points will be displayed as the same actual screen dot.

An Apple II Hi-Res screen dot occupies an area two x-coordinate values wide by two-and-two-thirds y-coordinate values high. This information is sometimes useful when you want to know which graphics locations are occupied by text characters, or how much to increase the x-coordinate to skip over three screen dots.

Text characters occupy an area seven Apple II Hi-Res screen dots wide by eight screen dots high, or 7*2=14 x-coordinate values wide by 8*(8/3)=21and1/3 y-coordinate values high. The chapter Computation Instructions, and the section on the AP: instruction in this chapter, both show examples that use this text character information in order to draw graphics which remain outside a set text window. Experimenting in the Graphics Editor, while watching the graphics and text positions displayed by the Graphics Editor's What-information command, may also help you to understand this material. Again, text-character positioning is normally handled for you automatically by Apple PILOT.

Text characters displayed by the Graphics T or X commands always appear in the usual white dots on a small, black background. Text characters displayed by the Type instruction (T:) usually appear the same way, but you can change to one of two other "screen modes" for other kinds of Type character displays. See the appendix ASCII Character Codes for a description of screen modes.

Color adds still another complication. Certain colors can appear only at odd horizontal screen-dot positions, while others can appear only at even horizontal screen-dot positions. Lines drawn in many colors

often appear to be more than one screen-dot wide. Other horizontal screen dots may also be affected when you plot colors near them. See the section on Color for more information.

TEXT SCREEN

The usual T: instruction can display characters on the screen in positions determined by a grid of 4Ø columns by 24 rows. The columns are numbered from Ø (the leftmost column) through 39 (the rightmost column). The rows are numbered from Ø (the topmost row) through 23 (the bottom row).

You will need to know these text-character positions if you change the "window" for text display, using the Graphics instruction's Viewport command (discussed later in this section). By default, the text window is the entire 24-row by 4Ø-column screen. It is often convenient to change this, however. For example, you might wish to draw graphics in the upper half of the screen and display text in a window or viewport which occupies the lower half of the screen.

Once you have set a text Viewport, all T: instructions will display text only at the character positions which fall within the specified window. For the purposes of the T: instruction, it is as if the normal text screen had shrunk to the size of the new viewport. If a T: instruction tries to display a line of text which extends beyond the width of the viewport, the line is broken when it reaches the right edge of the viewport and is continued on the next line. Lines are broken at the space after the last word which fits in the viewport on the current line. If a word is longer than the viewport line, the word is broken to fit on two lines. Re-formatting of your Type instruction's displayed text is done for you automatically, to fit in any size viewport. If the bottom line of the viewport already contains text, the next Type instruction causes the text (and graphics, if any) in the viewport to "scroll" up, making room at the bottom for the new line, while the old top line in the viewport disappears.

This Type instruction:

```
T:Chameleons are red,
: chameleons are blue;
```

normally causes this display:

```
Chameleons are red, chameleons are blue;
```

But setting the viewport to only 1Ø characters wide:

```
G: VØ,9,Ø,23
T:Chameleons are red,
: chameleons are blue;
```

causes this display:

```
Chameleons
are red,
chameleons
are blue;
```

However, do not confuse the T: instruction with the T command in the G: instruction. An instruction such as G:Tccc also displays characters on the screen, using the same character set, but those characters can begin at any graphics position and are not limited to the normal text-character positions or to the viewport.

TEXT AND GRAPHICS CURSORS

You are already familiar with the text cursor: it is a pointer, not always visible on the screen, which determines where the next character displayed by a T: instruction will appear on the screen. Each time another character is displayed by the T: instruction, the cursor moves to where the next character will appear.

There is also a graphics cursor, with a similar function: it is an invisible pointer which determines where the next graphics point will b plotted. This point is usually on the screen, but its position is not limited to the screen's boundaries. The graphics cursor can be moved only by graphics commands. Most graphics commands start their action a the current graphics cursor position, and leave the cursor at the location where that action ends. For example, the instruction

```
G: D2Ø,35
```

draws a line which starts at the current graphics cursor position and ends at the position x=2Ø,y=35 (relative to the Offset position). After this instruction, the graphics cursor position is the same as the position of the line's last point.

Full Screen (No Viewport)

When no viewport has been set, so that the entire screen is available for text, the text cursor follows any movement of the graphics cursor as long as the graphics cursor does not go beyond the boundaries of the screen. If a G: instruction draws a line to row 53, column 97 of the graphics screen, both cursors end up at row 53, column 97 (the text cursor is placed on the normal text-character position which includes the position of the graphics cursor). This lets you use the T: instruction to type labels at appropriate places in your graphics image.

If the graphics cursor is placed at a point which is beyond the boundaries of the screen, the text cursor assumes the screen position whose coordinates are each closest to the corresponding coordinate of the graphics cursor. Thus, if you Draw a line to the point 3ØØ,5ØØØ the graphics cursor ends up at 3ØØ,5ØØØ , but the text cursor is place

at the text character position which includes the graphics screen position 3ØØ,511 .

Note 1: the text cursor position is adjusted to match a new graphics cursor position only at the very END of any Graphics instruction which moves the graphics cusor. This is normally not a problem, but if you use a single Graphics instruction to move the graphics cursor and send a screen control character which refers to the text cursor's position, that screen control character takes its effect using the text cursor's OLD position, not the new graphics position. Just break the Graphics instruction into two parts, ending the first instruction after the cursor move and before sending the screen-control character. This will move the text cursor to the new graphics cursor position at the end of the first Graphics instruction.

Note 2: the text cursor is NOT moved to the graphics cursor position by Graphics instructions which do not move the graphics cursor. This means that you can display some text on the screen, using the Type instruction (which moves the text cursor), and then issue a G:X8,8,8,8 instruction, for instance, to backspace the text cursor. This Graphics instruction does NOT move the text cursor to the graphics cursor position (wherever that might be on the screen); it just moves the text cursor four characters back.

Note 3: the graphics cursor does NOT follow the text cursor. After a T: instruction displays a line of text on the screen, the graphics cursor remains just where it was before the T: instruction.

With a Viewport

When you use the Graphics instruction's Viewport command to set a window for text display, however, the behavior of the text cursor changes. First of all, the text cursor is now limited to those text character positions that lie within the viewport. Further, when a G: instruction moves the graphics cursor to a new location, the text cursor now remains exactly where it was.

Once you have used the Viewport command to set a viewport that does not include the entire screen, the text cursor and the graphics cursor are maintained completely independently. T: instructions move the text cursor (within the viewport) and G: instructions move the graphics cursor (anywhere on the screen, including the text viewport).

This lets you maintain a scrolling text dialog in the text viewport, while concurrently drawing and changing a graphics image in any other part of the screen. One advantage of this is that your graphics image (if it is outside the text viewport) does not scroll up and off the screen along with the text. Since the normal T: instruction can now display text only in the text viewport, you must use the Graphics instruction's T command to place text labels in your graphics image. Remember, though, that Graphics commands do not respect the text viewport, so a command such as G:ESØ erases the entire screen, including the text viewport. The only exception to this is the instruction G:ES (with no color number after the ES command), which erases only the text viewport.

Either of these equivalent instructions

```
G: VØ,39,Ø,23
```

or

```
G: V
```

re-establishes the full-screen text viewport, after which moving the graphics cursor once again drags the text cursor with it.

COLOR

Graphics colors are selected by number in the graphics commands Cn and ESn , where n is a numeric digit from Ø through 9, or a simple numeric variable whose value is any number from Ø through 9. Actually, any integer from Ø through 32767 may be used, or a numeric variable whose value is any number in the same range, but numbers greater than 9 are ignored, without changing the color previously set. The colors corresponding to the first 1Ø integers are as follows:

Color Number	Color Name	Color Number	Color Name	Color Number	Color Name
Ø	Black1	4	Black2	8	Reverse
1	Green	5	Orange	9	White
2	Violet	6	Blue		
3	White1	7	White2	(R and Q)	(Black)

The grouping of these colors is significant. There are many strange interactions between the method color monitors use to produce their colors, and the technique the Apple uses to store so much color information into so little memory. You may notice various odd effects when you draw any non-horizontal line across a colored background: black and white lines may appear as dashed or incomplete, colored lines may appear with black and white zebra stripes, or a jagged, colored "shadow" may appear around a line. To minimize these normal phenomena, use colors Black1 and White1 with Green and Violet, but use colors Black2 and White2 with Orange and Blue.

If you Plot a point using colors Ø, 3, 4, or 7 , the display of that point occupies two horizontal Apple II Hi-Res screen dots, to let the color show. Non-horizontal lines which are Drawn in colors Ø through 7 often appear wider than one screen dot. The colors Reverse (8) and White (9) can be used to draw finer lines which are only one screen-dot wide. For black-and-white monitors, you may wish to use mostly colors Ø, 8, and 9 for your drawings.

There is also a color Black which can be used to draw black lines and dots that are only one screen-dot wide, but it is not selected by any of the color numbers. Instead, the Graphics command Q always plots a point in the color single-dot Black, and the command R always draws a

line in the color single-dot Black, no matter which plotting color's number you have selected.

Reverse (8) is not actually a color, itself. A line drawn using Reverse takes its color from whatever the line passes over: it changes White1 to Black1 and Black1 to White1, White2 to Black2 and Black2 to White2, one-dot White to one-dot Black and one-dot Black to one-dot White, Green to Violet and Violet to Green, Orange to Blue and Blue to Orange. Reverse is a rather magical "color". It lets you draw a line, say, across a complex background, so that every part of the line shows up. When the Erase Screen command uses the color Reverse, it changes the entire foreground and background coloring of a picture without disturbing the picture itself.

The drawing color is set by default to White 1 (number 3), but if you change it with a G:Cn instruction it remains set to your color after that. The color used for erasing the screen in a G:ESn command is independent of the color used for drawing. An instruction such as G:ES (with no color number after the ES command) does not erase using a default color of black. Rather, it clears the text viewport portion of the screen without using any graphics routine at all.

When Graphics commands place text characters on the screen, they use the same character set used by the T: instruction. Regardless of what text-screen mode the Type instruction is using, or what plotting or background color you are currently using for graphics, the Graphics T and X commands always display their characters in the usual white, with each character on its own small, black background.

MODIFIER

The eXecute modifier may be used with the Graphics instruction to place entire stored graphics images on the screen. For more details, see the next section in this chapter, on the GX: instruction.

EXAMPLE

```
R: This draws a box:
G: ES4                  (Note to the author.)
G: C6                   (Erase Screen in color Black2.)
G: M3Ø,1Ø               (Choose drawing color Blue.)
C: D3Ø,11Ø              (Move to the point  x=3Ø,y=1Ø .)

G: D8Ø,11Ø              (Draw blue line from  x=3Ø,y=1Ø
                          to the point  x=3Ø,y=11Ø .)

G: D8Ø,1Ø               (Draw blue line from  x=3Ø,y=11Ø
                          to the point  x=8Ø,y=11Ø .)

G: D3Ø,1Ø               (Draw blue line from x=8Ø,y=11Ø
                          to the point  x=8Ø,y=1Ø .)

                        (Draw blue line from  x=8Ø,y=1Ø
                          to the point  x=3Ø,y=1Ø .)

R: This draws 2 large 4's   (Note to the Author.)
D: A4$(5Ø)                  (Reserve space in memory to
                              store up to 5Ø characters
                              in string variable  A4$ .)
C: A4$="G:M5Ø,Ø;D5Ø,1ØØ;D2Ø,3Ø;D7Ø,3Ø"  (Store the quoted string,
                                          a Graphics instruction
                                          which draws a figure 4,
                                          in variable  A4$ .)
G: O 1Ø,1Ø                  (Set the reference Offset
                              to the point x=1Ø,y=1Ø .)
G: C5                       (Choose drawing color Orange.)
XI: A4$                     (Execute the contents of  A4$
                              as a PILOT instruction.  This
                              draws an orange figure  4
                              with its base at x=6Ø,y=1Ø .)
T: FOUR                     (Display this text starting at
                              the graphics cursor position,
                              next to the last figure  4 .)
G: O 2ØØ,45Ø                (Set the reference Offset
                              to the point x=2ØØ,y=45Ø .)
G: C7                       (Choose drawing color White2.)
XI: A4$                     (Execute the contents of  A4$
                              as a PILOT instruction.  This
                              draws a white figure  4
                              with its base at x=25Ø,y=45Ø.)
G: V2Ø,26,5,15              (Establish a text window from
                              text column 2Ø to column 26,
                              and from row 5 to row 15.)
T:'Twas brillig, and the slithy   (Display this text in the text
:toves did gyre and gimble in the   window.)
:wabe:  all mimsy were the boro-
:goves, and the mome raths outgrabe!
G: Tfour                    (Display this text starting at
                              the graphics cursor position,
                              which has remained next to
                              the last figure  4 .)
G: ES8                      (Reverse all the colors on
                              the screen.)
G: ES8                      (Reverse again, back to the
                              original colors.)
```

Note that the seven G: instructions which draw the box, at the beginning of this example, could also be written as a single G: instruction with a long list of commands. It would look like this, in the Lesson Text Editor:

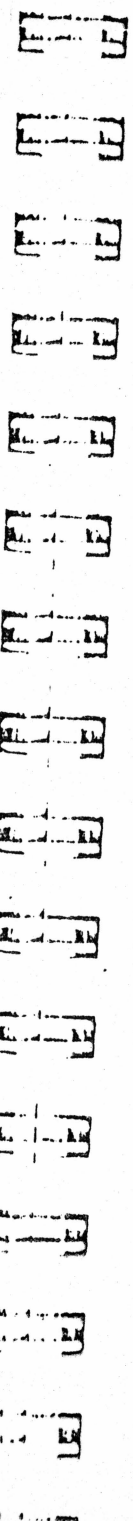
```
G:ES1;C6;M5Ø,5Ø;D5Ø,1ØØ;D1ØØ,1ØØ;D1ØØ,5
:Ø;D5Ø,5Ø
```

The single, long G: instruction will execute much more quickly than the multiple, shorter G: instructions because the graphics routines only need to be read into the Apple from diskette once.

In this example, the G: instruction to draw a figure "4" was stored in a string variable and then executed by an XI: instruction. This figure-drawing G: instruction could also have been written as a subroutine and then executed by a U: instruction.

Before the Viewport is set, the text cursor follows the graphics cursor, and the T: instruction displays text starting where the last graphics command ended. After setting the Viewport, the T: instruction displays its text within the Viewport, independent of any graphics moves. Note how the text in the Viewport scrolls without affecting the graphics on the rest of the screen. Any graphic element that happens to be within the Viewport, however, will be scrolled away right along with the text.

For another graphics example, see the chapter Computation Instructions, at the end of the section called Using Compute for Assignment.



GX: EXECUTE GRAPHICS FILE

GX: file name of a graphics image stored on the lesson diskette

The eXecute Graphics instruction reads a graphics file stored on the lesson diskette, and re-creates the stored graphics image on the student's screen. Unlike the G: instruction, no graphics commands appear in the object field of the GX: instruction. Instead, the file name of the stored graphics image is the object. For example, to use the stored graphics image whose file name is PICTURE, your lesson would use this instruction:

```
GX: PICTURE          (Find the graphics image whose file
                      name is PICTURE , stored on the
                      lesson diskette, and re-draw it
                      step by step on the screen.)
```

The graphics image file is created and named while you are in the Graphics Editor. If you are using a normal, one-disk-drive Lesson Mode system, all image files used by GX: instructions must be on the same diskette as the lesson which includes the GX: instructions. For using a system with additional disk drives, see the appendix Using More Disk Drives.

If a graphics file with the specified name is not found on the lesson diskette, you are given no message, even if you are running a lesson in Author mode, and the Error Flag is not raised. Lesson execution simply goes on to the next instruction. This lets you write and test lessons which include GX: instructions, before you develop the stored graphics images named in the GX: instructions. Later, when you provide the correctly named image files, your lesson will automatically incorporate those images.

NORMAL GRAPHICS FILES

A GX: instruction which specifies a normal file name (such as PICTURE) re-creates the stored image step-by-step, just as it was created in the Graphics Editor. The previous contents of the screen are not removed by this instruction. The stored graphics image will be drawn over any text or graphics already on the screen unless a G:ES (erase screen) instruction precedes the GX: instruction.

QUICK-DRAW GRAPHICS FILES

It is also possible, when you are in the Graphics Editor, to give your graphics image a file name with an exclamation point anywhere in the name. For example, you might give the name SUNSET! to a drawing of the sunset. Such a file is called a "quick-draw" file, and the Graphics Editor saves the color of every screen dot for that image, in addition to the sequence of steps which created that image. A GX: instruction such as

```
GX: SUNSET!                    (Find the graphics image whose file
                                 name is  SUNSET! , stored on the
                                 lesson diskette, and quickly flash
                                 that complete image on the screen
                                 all at once.)
```

specifies the name of a quick-draw graphics file, and quickly replaces the entire contents of the student's screen with the stored image. The stored quick-draw image is not re-drawn step by step, but is flashed on the screen all at once, erasing anything that was on the screen before the instruction. The penalty for this speed is that it takes much more space on the lesson diskette to store a quick-draw file than to store an ordinary graphics file.

Unlike a normal, step-by-step file, the quick-draw image cannot be restricted to certain portions of the screen. A quick-draw file always replaces everything on the student's screen, including any text, even in a viewport.

If the dot-by-dot quick-draw image file is not found on the diskette when a quick-draw file name is specified, PILOT looks for the step-by-step command file with the same file name. Lesson execution goes on to the next instruction without affecting the screen only if neither portion of your quick-draw file can be found. [Note: if you have access to an Apple Pascal system, you can look at the directory listing of your lesson diskette graphics files in the Pascal Filer. A dot-image file has the suffix .FOTO after its file name, while a command-sequence file has the suffix .GRAF . The quick-draw file SUNSET! appears as two files: SUNSET!.GRAF and SUNSET!.FOTO .]

See the section of the Apple PILOT Editors Reference Manual dealing with the Graphics Editor for further details on creating and using the graphics images.

EXAMPLE

```
G: ES5;MØ,Ø                    (Erase the Screen to the
                                 background color Orange,
                                 and move graphics and text
                                 cursors to lower left corner.)
T:          This is an Apple.  (Display this text at the
                                 bottom of the screen.)
CX: APPLE                      (Re-draw the stored graphics
                                 image named Apple, step by
                                 step on the screen.)
W: 1Ø                          (Wait 1Ø seconds, or until
                                 student presses any key.)
GX: GLOBE!                     (Quickly replace current screen
                                 contents with the quick-draw
                                 graphics image named  GLOBE! .)
G: MØ,Ø;TWhat is this a map of? (Move graphics and text cursors
                                 to lower left corner of the
                                 screen, and display text.)
```



```
A:                             (Accept student's response.)
M:  WORLD!EARTH!PLANET!GLOBE   (Did student type one of these
                                 match words?)
...                              ...
...                              ...
```

AP: ACCEPT POINT

AP:

The Accept Point instruction waits for the student to indicate a point on the screen, using the game controls. It then stores the indicated screen location for use by later instructions. If any variable names appear in the object field of this instruction, they are ignored.

When the AP: instruction is executed, a pair of intersecting "cross-hairs" appears on the screen. By using the game controls, the student can move the intersection of the cross-hairs to any point on the screen. Game control Ø sets the horizontal position, and game control 1 sets the vertical position. When the cross-hairs are centered on the desired location, the student presses the button on either game control to save the graphics coordinates of the indicated screen position. The horizontal coordinate (a number from Ø through 559) is stored in system variable %X , and the vertical coordinate (a number from Ø through 511) is stored in system variable %Y .

These system variables may be used in other instructions to let the student point to various words or graphics items on the screen, to let students do their own graphics, etc. For example, you could Accept a Point and then Draw a line to that point, over and over, using instructions such as these:

```
C: ESØ;C9          (Clear the screen to Black1;
                    set drawing color to White.)
AP:                (Display cross-hairs; Accept
                    the coordinates of the cross-
                    hairs location when student
                    presses game-control button.)
C: H = %X          (Store x-coordinate %X in
                    simple numeric variable H .)
C: V = %Y          (Store y-coordinate %Y in
                    simple numeric variable V .)
G: DH,V            (Draw a White line to the point
                    x=H,y=V .)
J: @A              (Jump back to last Accept; in
                    this case, the AP: .)
```

System variables are discussed in the chapter Computation Instructions.

RESPONSE TIMING

If the last Accept instruction was an AP: instruction, the TIM(Ø) function returns the amount of time the AP: instruction waited before the student pressed a game-control button to Accept the coordinates of the cross-hairs location. For example, you could display the student's response time with these instructions:

```
AP:                (Accept cross-hairs location
                    when student presses a
                    game-control button.)
C: T2 = TIM(Ø)     (Store the student's response
                    time in simple numeric
                    variable T2 .)
T: You took #T2 seconds
: to choose your point.
                   (Display this text, substituti
                    the response time T2 for
                    the name #T2 .)
```

The TIM function is described in the chapter Advanced Programming.

Normally, the Accept Point instruction will wait indefinitely for the student to adjust the cross-hairs. Program execution continues after the student presses either game-control button to Accept the coordinate of the indicated screen location. However, if the Tn response-timing option has been set by the last PR: instruction, the AP: instructio waits for a maximum of n seconds (where n is an integer from Ø through 32767). If the student has not pressed a game control button after n seconds, program execution continues just as if the button ha been pressed. In this case, the coordinates Accepted indicate where the cross-hairs were placed when the time ran out.

For example, to set a maximum response time of 1Ø seconds (at the same time terminating any other PRoblem control options), your lesson could use this instruction:

```
PR: T1Ø            (Set maximum response time for
                    future Accept and Accept Poi
                    instructions to 1Ø seconds.)
```

See the discussion of the PRoblem instruction, in the chapter Text and Response Instructions, for details about setting the response-timing control option.

EXAMPLE

```
C: R = 5           (Store number 5 in simple
                    numeric variable R .)
PR: U TR           (Convert response letters to
                    capitals; set maximum
                    response time to R , in
                    this case 5 seconds.)
G: VØ,39,Ø,1;ES;C8 (Allow text only in top two
                    lines; Erase screen to
                    black; set plotting color
                    to Reverse.)
*START             (Label for this section.)
C: H2 = RND(56Ø)   (Store a randomly-selected
                    horizontal coordinate in
                    numeric variable H2 .)
C: V2 = RND(465)   (Store a randomly-selected
                    vertical coordinate in
                    numeric variable V2 .)
G: PH2,V2          (Plot a dot at screen point
                    x=H2,y=V2 .)
```

```
T:You have 5 seconds to catch the dot.   (Display this text.)
AP:                                      (Accept the coordinates of
                                           the screen point under
                                           the cross-hairs.)
S(ABS(H2-%X)>2 ! ABS(V2-%Y)>3): 1,3ØØ    (If the Accepted point misses
                                           the Plotted point by more
                                           than one screen-dot, play
                                           a low tone.)
TC:Sorry, you missed it.                 (If same condition is true,
                                           display this text.)
JC: REPLAY                               (If same condition is true,
                                           Jump to label  REPLAY .)
S:3Ø,7;35,7;39,7;42,25;Ø,7;39,7;42,15Ø   (The Accepted coordinates were
                                           both within 2 of the Plotted
                                           coordinates, so play tune.)
C: T = TIM(Ø)                            (Store the response time in
                                           numeric variable  T .)
T:Good!  You got it in #T  seconds.      (Display text, substituting
                                           response time  T for  #T
                                           and one space .)
*REPLAY                                  (Label for this section.)
TH:Want to try again?                    (Display this text, leaving
                                           cursor at end of the text.)
A:                                       (Accept student's response.)
M: %NO!%N%!NEVER!FORBID!KIDDING          (Did student type any of
                                           these negative answers?)
EY:                                      (If so, end this lesson.)
G: PH2,V2                                (Erase dot by plotting point
                                           at x=H2,y=V2 in color
                                           Reverse for the 2nd time.)
J: START                                 (Jump back to label  START .)
```

This example is a game of "Catch the Dot". It shows a white dot on the screen at a random location, then displays the Accept Point cross-hairs. The student has a maximum of five seconds to center the cross-hairs on the dot, using the game controls. The game continues when the student presses either game-control button, or when 5 seconds is up, whichever comes first.

The coordinates of the Accepted point are then compared with the coordinates of the dot. The student has succeeded in catching the dot if the Accepted x-coordinate is within plus or minus 2 of the dot's x-coordinate, and the Accepted y-coordinate is within plus or minus 3 of the dot's y-coordinate. Since a screen dot occupies an area two x-coordinate values wide by two-and-two-thirds y-coordinate values high, this test lets the student catch the dot if the Accepted point is within one screen dot of the "target" dot, in any direction.

Note that Plotting in the color Reverse first reverses a black background dot to white. Plotting the same point again reverses the white dot back to black, effectively erasing the dot.

The randomly-selected y-coordinate for the "target" dot has been limited to the range from Ø through 464. This ensures that the dot will not be placed in the text viewport, as you will see. The viewport allows two lines of text at the top of the screen, and a text character is 7 screen-dots wide by 8 screen-dots high. Each screen dot occupies an area two x-coordinate values wide by two-and-two-thirds y-coordinate values high. Therefore, the two rows of viewport text occupy an area 16 screen-dots high (2*8), or about 43 y-coordinate values high (16*2.67). We subtract 43 from 511, the screen's top y-coordinate, to calculate the highest allowable y-coordinate of 468 , and then subtract another 4 from that to make sure the "target" dot never actually touches the bottom of the text.

TX: EXECUTE CHARACTER-SET FILE

TX: no object, or file name of a character set stored on lesson diskette

The eXecute character seT instruction replaces the usual set of text character display images by a set of images stored under the specified file name, on the lesson diskette. Following this instruction, any text character typed by the student or displayed by the program will cause one of the images from the new character set, rather than the usual image, to appear on the screen. This instruction lets your lessons use alphabets other than the standard alphabet. To return to the standard character set, use a TX: instruction with no object.

Unlike the usual T: instruction, the TX: instruction's object field does not contain text to be displayed on the student's screen. Instead, the file name of the stored character set is the object. For example, to begin using the stored character set whose file name is GREEK , your lesson would use this instruction:

```
TX: GREEK                    (Replace the old set of character images
                              with the character set whose file name
                              is GREEK , stored on the lesson
                              diskette, and use the new images for
                              future character displays.)
```

The character set file is created and named while you are in the Character Set Editor. If you are using the normal, one-disk-drive Lesson Mode system, all character set files used by TX: instructions must be on the same diskette as the lesson which includes the TX: instructions. For using systems with additional disk drives, see the appendix Using More Disk Drives.

If no character set file with the specified name is found on the lesson diskette, no message is given, even if you are running a lesson in Author mode, and the Error Flag is not raised. The current character set is not changed in such a case, and execution of the lesson continues with the next instruction. This lets you write and test lessons which include TX: instructions, before you develop the stored character sets named in the TX: instructions. Later, when you provide the correctly named character set files, your lesson will automatically incorporate those character sets.

When you are finished using the new character set, and want to return to the standard character set, use this instruction:

```
TX:                          (Resume using the standard character set
                              for future character displays.)
```

Three other situations automatically return your lessons to using the standard character set:

1. Every time Apple PILOT is started or re-started, the standard character set is invoked.

2. If a special character set is in use when an LX: instruction is executed, the standard character set is re-invoked. The un-modified L: instruction does NOT change the character set being used.

3. When a lesson is running in Author Mode and a special character set is being used, the standard character set is re-invoked in order to display any error message. The lesson does NOT automatically return to the special character set following the error message.

Suppose the GREEK character set has been designed so that a drawing of greek "alpha" corresponds to the usual drawing for "A". After your lesson executes the instruction TX:GREEK , an alpha will appear on the screen every time the student presses the A key and every time the T: or G:T instructions specify an A as the character to display.

The images stored in the character set file may be any drawings at all; they are not necessarily restricted to drawings of alphabetic characters. For instance, you might design several images to look like a person in various stages of walking. A sequence of graphics G:T instructions, using the characters associated with your images, could then create the appearance of a person walking across the screen.

It is also possible to create images, each of which is a portion of a larger drawing. Clumps of all the characters associated with these images will then produce the entire drawing on the screen. For example, the character set named MAXWELL, used by the LEGENDS demonstration lesson, consists of character-images which, in certain groups of five characters, form images of a person in various positions.

The new character set only changes the drawings associated with screen displays of characters, and only when the student is running the lesson Internally, the Apple continues to use the standard characters, just as the keys themselves do not change (unless you change the labels on them). Thus, when you are typing instructions in the Lesson Text Edito you will always use the standard ASCII (English) letters, even if you know that they will later appear in some other form when they are displayed on the student's screen.

See the portion of the Apple PILOT Editors Manual dealing with the Character Set Editor for further details on creating and using stored character sets.

EXAMPLE

```
PR: U                        (Convert all response letters
                              to upper-case.)
TX: JIBBERISH                (Replace the old set of charact
                              images with the character set
                              whose file name is JIBBERISH
                              stored on the lesson diskette
                              and use the new images for
                              future character displays.)
```

```
*START                                  (Label for this section.)
T: Skveid endbr8foØ dks ksie            (Display the JIBBERISH image
: *kej8jsm vhhifke 2hdn3 j sjak?         corresponding to each of
                                         these characters.)
A:                                      (Accept student's response,
                                         displaying the JIBBERISH
                                         image corresponding to each
                                         key the student presses.)
M: KRIEDF!BUROPDS!ERSD!T*3BF            (Did the actual keys pressed
                                         by the student include any
                                         of these four sequences?)
TY: Qwebghe!                            (If Yes, display the JIBBERISH
                                         images corresponding to these
                                         characters.)
JY: NEXT                                (If Yes, Jump to label NEXT .)
T: Gewnt.  Towo andkei.                 (No, so display the JIBBERISH
                                         images corresponding to these
                                         characters.)
J: START                                (Jump back to label START .)
*NEXT                                   (Label for this section.)
TX:                                     (Resume using the standard
                                         character set for future
                                         character displays.)
T: Now we will try the same thing       (Display the standard images
: again, this time in English.           corresponding to these
                                         characters.)
...                                      ...
...                                      ...
```

The file named JIBBERISH was created earlier, in the Character Set Editor, and stored on the lesson diskette. Each standard symbol typed in the Lesson Text Editor has a corresponding drawing or character image in the JIBBERISH file. After the instruction TX:JIBBERISH any character displayed by a T: instruction or typed by the student will appear on the screen, not in its usual form, but as the JIBBERISH image which corresponds to that character.

To write the lesson, the author must have a table telling which standard character corresponds to each JIBBERISH image. To Type or Match a given series of JIBBERISH images, the author then uses the corresponding series of standard characters in a T: or M: instruction, knowing that the appropriate transliteration will take place when the lesson is running.

S: SOUND

S: pitch of note, duration of note
or
S: pitch,duration; pitch,duration; pitch,duration

The Sound instruction plays a musical note or notes over the Apple's speaker. You can use this sound to get the student's attention, to signal a wrong answer (or a right one), or to dramatize a visual effect. You can even use the Sound instruction to play simple tunes. For longer, more complicated musical pieces or sound effects, see the SX: instruction.

To play a note, the Sound instruction's object field must specify how high or low a note to play (that is, its "pitch"), and then how long to play that note (its "duration"). For example, to play a middle C that lasts about half a second, you could use this instruction:

```
S: 8,1ØØ                                (Play a note of pitch 8 and
                                          duration 1ØØ over the
                                          Apple's built in speaker.)
```

The note's pitch should be an integer from Ø through 5Ø . Pitch Ø makes no sound at all; it is used to specify a rest or silence. Pitches 1 through 5Ø yield a rising tempered (approximately) chromatic scale, in which each pitch number gives a note which is one half-step higher than the previous pitch number. A note of pitch 8 is approximately middle C. There are 12 half-steps in an octave; so pitch 2Ø is a C an octave above middle C, pitch 32 is a still higher C, and pitch 44 is the highest C you can play.

A note's duration should be an integer from Ø (the shortest note) through 255 (the longest). A duration of Ø specifies a note which is so short it sounds almost like a "click", especially with lower pitch numbers. A note of duration 2ØØ lasts about a second.

Actually, any integer from Ø through 32767 may be used for pitch and duration. Pitch numbers higher than 5Ø are converted to pitch 5Ø; duration numbers higher than 255 are changed to duration 255. The name of a simple numeric variable may appear in place of any pitch or duration number. The number stored in this variable need not be an integer, but it must be within the range limits imposed on pitch and duration. A non-integer variable value is not rounded, it is truncated: any portion of the number to the right of the decimal point is simply discarded.

Several notes may be specified in a single S: instruction by listing their pitch,duration pairs in sequence, separated by semicolons. For example, this instruction plays a C scale:

```
S:8,1ØØ; 1Ø,1ØØ; 12,1ØØ; 13,1ØØ;          (Play eight notes over
:15,1ØØ; 17,1ØØ; 19,1ØØ; 2Ø,1ØØ             the Apple's speaker.)
```

The spaces are not necessary; they just make the instruction easier to read.

When you are first developing a sound effect, it is usually better to use several S: instructions with just one or a few object notes each, rather than one S: instruction with a long list of notes. Changing individual notes in the sequence is much easier when those notes are not buried in a long list of notes.

Later, when you are satisfied with your tune, you can always combine the separate instructions into one instruction with a long note-list. Combining separate S: instructions into one may make the program a little more compact and easier to read in the Lesson Text Editor, but it has little effect on the sounds produced. At most, you will notice a very slight decrease in the "space" between successive notes after combining them into one instruction.

If you do type a long Sound instruction in the Lesson Text Editor, the Editor automatically breaks your instruction each time the line exceeds 39 characters, and continues it (beginning with a continuation colon) on the next line. You may also break the instruction anywhere you wish, by pressing the RETURN key. If you then wish to continue the Sound instruction, you must begin the new line with a continuation-colon or a new S: instruction.

Even though the Editor may break your instruction in the middle of a note specification when it reaches the thirty-ninth character, the instruction will be executed correctly. When the instruction is executed, any breaks and continuation-colons are ignored, whether they were introduced by you or by the Editor. The object text is treated as if it were one long, unbroken string. This means that if you break a Sound instruction, and then colon-continue the list of notes on the next line, you must not leave out any separating semi-colons or introduce any illegal spaces in the middle of a number.

Regardless of the number of lines a single S: instruction occupies, only the first 25Ø characters of the instruction are actually used at execution time. The continuation-colons are not counted in this total. If the instruction is longer than 25Ø characters, any characters after the 25Øth character are simply ignored.

MODIFIER

The Sound instruction can be used with the eXecute modifier to play musical compositions and sound effects created in the Sound-Effects Editor. For example, your lesson could play the sound-effect stored in the diskette file named DRUMROLL by using this instruction:

```
SX: DRUMROLL                (Find the sound effect whose file
                              name is DRUMROLL , stored on the
                              lessons diskette, and play it over
                              the Apple's built in speaker.)
```


For more information, see the next section in this chapter, on the SX: instruction.

EXAMPLE

```
PR: U                           (Convert all response letters
                                  to capital letters.)
C: N = 1                        (Set value of variable N to 1.
T: Wait until you hear a "beep";(Display these lines
: then quickly type the names of  of text.)
: five famous women.
*NAMES                          (Label for this section.)
W: 2                            (Wait 2 seconds or until any
                                  key is pressed.)
S: 3Ø,1Ø                        (Play a short "beep" on the
                                  Apple's speaker.)
A:                              (Accept student's response.)
S: 2Ø,2Ø                        (Play a lower, longer "beep".)
C: T = TIM(Ø)                   (Store the response time in
                                  simple numeric variable T .)
T: It took you ØT  seconds      (Display these lines of text,
: to name them.                   substituting the response
                                  time T for ØT .)
J(N=2): NEXT                    (If N=2, Jump to label  NEXT
T: Do you think you could name  (N=1, so display this text.)
: five famous men in less time?
A:                              (Accept student's response.)
T: Well, give it a try, when    (Display this text.)
: you hear the "beep".
C: N = 2                        (Set value of variable N to 2
J: NAMES                        (Jump back to label  NAMES .)
*NEXT                           (Label for this section.)
T: Was it easier to think of    (Display these lines
: famous women or famous men?     of text.)
A:                              (Accept student's response.)
T: I see.  Think about the reasons (Display this text.)
: for this, while you listen to
: the following tune.
S: Ø,255                        (Short silence.)
S: 8,5Ø; 11,5Ø; 8,1Ø5           (Plays tune of "Five foot
S: 8,5Ø; 12,5Ø; 8,1Ø5             two, eyes of blue..." .)
S: 8,5Ø; 13,5Ø; 8,5Ø; 13,5Ø
S: 8,25; 13,75; 8,47; Ø,3
S: 13,5Ø; 16,5Ø; 13,5Ø; 16,25; 13,75
S: 11,1Ø5; 13,11Ø; 16,22Ø

T: Do you think this describes  (Display this text.)
: a famous woman?
...                             ...
...                             ...
```

CHAPTER 7

FILE HANDLING INSTRUCTIONS

136 GENERAL FILE INFORMATION
136 Files
136 Records
136 Opening a File
137 Closing a File

138 FOX: CREATE AND OPEN NEW FILE
139 The Maximum Record Number
139 Deleting a Data File
139 The String Variable
139 The New File
140 Example

142 FIX: OPEN EXISTING FILE
143 The Maximum Record Number
143 The String Variable
144 Example

146 FO: OUTPUT TO OPEN FILE
146 The Record Number
147 The String Variable
147 Example

151 FI: INPUT FROM OPEN FILE
151 The Record Number
152 The String Variable
152 Example

GENERAL FILE INFORMATION

FILES

Your lessons can easily store student responses, scores, or other information onto the lesson diskette, and then retrieve the same information later in the same lesson or in another lesson. Information is stored on the lesson diskette in a named "file", similar to the diskette files which store your lessons, graphics, character sets, and sound effects. You give each file a specific name, and refer to that file by its name from then on.

When you create a lesson, graphics, character-set, or sound-effects file using one of the special Editors, that file's name is added to the list which you see each time you select the appropriate Editor. However, the names of files created when a lesson is running, by instructions within that lesson, never appear on any Editor list. You must remember what files your lessons have created, or refer to the instructions which create those files, in the text of the lesson itself.

[Note: if you have access to an Apple Pascal system, you can use the Pascal Filer to list all the files on a PILOT lesson diskette. Each PILOT file appears in the Pascal Filer with a characteristic suffix added to the usual PILOT file name. Lesson files end in .TEXT , graphics files end in .GRAF or .FOTO , character-set files end in .FONT , sound-effects files end in .SONG , and files created by instructions in a running lesson end in .DATA .]

RECORDS

An individual item of information is saved in a file sub-unit called a "record". Each record occupies half of a block on the lesson diskette and may hold one string only, up to 255 characters long. A record may hold a word, a sentence, the string representation of a number, a PILOT instruction: anything that can be stored in a string variable. A number cannot be put in a file record directly. The number must first be converted to a string, and then that string can be saved in a file record.

Each record in a file has a "record number". The file's first record is number Ø, the second is record number 1, and so on. You can store any item into any of the file's records, in any order, and some records may remain empty. However, you must remember the record number of each item your lesson saves. When you retrieve that information later, your instructions will have to ask for each item by its record number.

OPENING A FILE

Before your lesson can store any item in a record, or retrieve the contents of any record, you must first tell Apple PILOT which file you are talking about. This process of announcing the file's name and preparing the file for storing or retrieving information is called "opening" the file.

PILOT has two different ways to open a file. The FOX: instruction creates and opens a new file on the lesson diskette; the FIX: instruction re-opens an existing file. Each time you open a file, your file-opening instruction specifies the maximum record number that your storage or retrieval instructions can use. You should remember this maximum number, and use it consistently every time you open the file.

Once you have opened a file, all of your subsequent storage (FO:) or retrieval (FI:) instructions are assumed to refer to records within that file. Only one file can be open at any time.

Note: Common PILOT uses the FO: instruction both to store information in an open file and to open a file when no file is currently open. In Apple PILOT, you must use the special file-opening instruction FOX: or FIX: to open a file.

CLOSING A FILE

Announcing that you are finished using the currently open file is called "closing" the file. In general, you must open a file yourself, but Apple PILOT closes the file for you, automatically. For example, any instruction which opens a new file automatically closes any currently open file first.

The following instructions and commands close any open file:

FOX: , FIX: , GX: , TX: , SX: , LX: ,

CTRL-C in response to A: , final E: , end of lesson without E:

Note that pressing the RESET key (or turning the Apple's power off) does NOT close any open file. Normally this causes no trouble unless your lesson was in the midst of storing information at that moment.

Closing a file just means you are temporarily through storing or retrieving information in that file. The file and its contents remain on the lesson diskette, but that information is not available until the file is re-opened. See the FOX: instruction for information on actually deleting a lesson-created file from a diskette.

FOX: CREATE AND OPEN NEW FILE

FOX: maximum record number, string variable containing name of new file
or
FOX: Ø, string variable containing name of file to be deleted

Technically, this instruction is a modified FO: instruction, but its use is quite different. The FOX: instruction first closes any file which is already open, and then creates and opens a new file on the lesson diskette, using the file name stored in the specified string variable. Subsequent storage and retrieval instructions (FO: and FI:) will refer to records within this file as long as it remains open; the record numbers specified by those instructions must not exceed the maximum record number given in the FOX: instruction.

For example, if you wanted your program to create and open a new file named COMMENTS , for storing and retrieving up to twenty-one records numbered Ø through 2Ø, you might use these instructions:

```
D: F$(1Ø)                (Reserve space in memory for
                          storing up to 1Ø characters
                          in string variable F$ .)
C: F$ = "COMMENTS"       (Store the file name COMMENTS
                          in string variable F$ .)
FOX: 2Ø,F$               (Create the new file COMMENTS;
                          and open it for storing and
                          retrieving information in
                          records numbered Ø through 2Ø.)
```

After these instructions, you could store a student's response in record number 12 of the file COMMENTS, using instructions such as

```
D: C$(8Ø)                (Reserve space in memory for
                          storing up to 8Ø characters
                          in string variable C$ .)
C: C$ = %B               (Assign the contents of the
                          Answer Buffer %B to
                          string variable C$ .)
FO: 12,C$                (Store the contents of string
                          variable C$ into record 12
                          of the currently open file.)
```

and retrieve the stored response later using this instruction:

```
FI: 12,C$                (Retrieve the contents of
                          record 12 in the currently
                          open file, and store in
                          string variable C$ .)
```

If you wish to re-open an existing file without creating any new file, see the FIX: instruction, discussed later in this chapter.

THE MAXIMUM RECORD NUMBER

When you use the FOX: instruction to create a new file, the maximum record number must be an integer from 1 through 32767, or a simple numeric variable containing any number in the same range. In practice, however, the maximum number of records is limited by the amount of spac available on the lesson diskette when the FOX: instruction is executed. There are a minimum of two records in a file, numbered Ø and 1 .

DELETING A DATA FILE

If a FOX: instruction specifies a maximum record number of Ø, this does not create a new, one-record file. Instead, Apple PILOT looks for an existing lesson-created file with the specified file name. If such file is found, the Apple DELETES that file from the diskette. This allows your lessons to remove from the diskette any files that will no longer be needed, before the lesson comes to an end. Also, any other lesson can use this instruction to delete a file after retrieving that file's stored information.

If the file specified for deletion is not found, no error message is given and the Error Flag is not raised. Lesson execution quietly goes on to the next instruction. This lets you attempt to delete a file BEFORE creating a new file with the same name (for instance), just in case there was an old file with that name. The usual process followed when creating a file is the opposite: FOX: creates the new file and THEN automatically deletes any old file with the same name.

THE STRING VARIABLE

The file name for a new file (or a file to be deleted) must be stored in a string variable, and only the string variable's name may appear in the FOX: instruction, not the file name itself. A file name must not exceed ten characters. The name should not contain any colons, semicolons, commas, pound-signs (#), dollar-signs, equals-signs, square brackets, asterisks, or spaces. It does not matter whether you type the file name in capital letters, lower-case letters, or both. Internally all file name letters are always converted to capital letters.

For specifying names of files on disk drives other than the usual default drive (the only drive in a one-drive Lesson Mode system, or the secondary drive in an Author Mode system), see the appendix Using More Disk Drives.

THE NEW FILE

When the FOX: instruction creates a new file, it reserves enough spac on the lesson diskette to store the specified number of records (from record Ø through the specified maximum record number). It then fills the entire storage area of the file with "null" characters (ASCII code Ø), erasing any old information that may have been stored there. From then on, the file always contains exactly the same number of records it contained when created, even if you actually store information in only one record.

Each diskette block can contain two records. Records Ø and 1 are in the first block of the file, records 2 and 3 are in the second block, and so on. However, note that the instruction FOX:Ø,X$ attempts to DELETE the file whose name is stored in X$, so that the smallest file you can actually create contains two records (record Ø and record 1).

If there is not enough room on the diskette for a file of the specified size, the file created occupies all of the largest unused area on the diskette. A line at the bottom of each Editor menu tells how many blocks are available at that time. Each block can contain two records. But note that FOX: instructions, encountered in any running of this lesson or other lessons on the same lesson diskette, may change the amount of diskette space available.

If a file with the specified file name already exists on the lesson diskette, the new file is first created in the remaining diskette space, and THEN the old file with that name is deleted. This leaves some unused blocks where the old file was, not at the end of the diskette. When the Editors create or change a file, they "compact" the diskette after storing the new file. This makes sure all the unused diskette blocks are in one place at the end of the diskette. The diskette is not "compacted", however, after a FOX: instruction creates a new file. Successive FOX: instructions creating files with the same file name may leave "pockets" of unused blocks here and there on the diskette. The total number of unused blocks may thus be much larger than the largest contiguous set of unused blocks which can store a file. This situation will be remedied the next time you perform an action which normally compacts the diskette.

In a normal, one-disk-drive Lesson Mode system, your lesson creates and uses files on the same diskette the lesson is stored on. For using systems with additional disk drives, see the appendix Using More Disk Drives.

EXAMPLE

```
PR: E                          (Turn on the Escape option.)
D: F$(1Ø)                      (Reserve space in memory for
                                storing up to 1Ø characters
                                in string variable F$ .)
D: R$(8Ø)                      (Reserve space in memory for
                                storing up to 8Ø characters
                                in string variable R$ .)
C: F$ = "COMMENTS"             (Store the file name COMMENTS
                                in string variable F$ .)
FOX: 2Ø,F$                     (Create the new file COMMENTS;
                                and open it for storing and
                                retrieving information in
                                records numbered Ø through 2Ø.)
C: N = Ø                       (Store the number Ø in simple
                                numeric variable N ; this
                                variable will contain the
                                record-number counter.)
J: START                       (Jumps over SYSX routine to
                                label START .)
*SYSX                          (Label for Escape routine; the
                                lesson Uses this routine if
                                any response begins with @ .)
C: R$ = %B                     (Assign the contents of the
                                Answer Buffer %B to
                                string variable R$ .)
FO: N,R$                       (Store the contents of string
                                variable R$ in record N
                                of the currently open file.)
C: N = N+1                     (Add one to the record-number
                                count stored in variable N .)
T(N>2Ø):No more room for comments.
                               (If N exceeds 2Ø, display text.)
PRC:                           (If N exceeds 2Ø, turn off the
                                Escape option.)
E: @A                          (End SYSX routine; jump back
                                to same Accept which initiated
                                this Escape routine.)
*START                         (Label for main body of lesson.)
...                            ...
...                            ...
```

You might use a lesson portion like this example at the beginning of a program which you are developing. It lets you store a comment at any response point, by typing an @ sign followed by the comment. It then returns you to the same response point to Accept your normal response and continue with the lesson. Note that this example creates a new file COMMENTS every time you run the lesson, deleting any previous COMMENTS file. You would need to write another lesson or lesson portion to retrieve the comments stored by this example. See the examples at the end of the other sections in this chapter for more uses of these instructions.

FIX: OPEN EXISTING FILE

FIX: maximum record number, string variable containing name of old file

Technically, this instruction is a modified FI: instruction, but its use is quite different. The FIX: instruction first closes any file which is already open, and then uses the file name stored in the specified string variable to open a file which already exists on the lesson diskette. The Error Flag is raised if the named file is not found on the lesson diskette. Subsequent storage and retrieval instructions (FO: and FI:) will refer to records within this file as long as it remains open; the record numbers specified by those instructions must not exceed the maximum record number given in the FIX: instruction.

For example, if you wanted your program to open an existing file named COMMENTS , for storing and retrieving up to twenty-one records numbered Ø through 2Ø, you might use these instructions:

```
D: F$(1Ø)                    (Reserve space in memory for
                               storing up to 1Ø characters
                               in string variable F$ .)
C: F$ = "COMMENTS"           (Store the file name COMMENTS
                               in string variable F$ .)
FIX: 2Ø,F$                   (Find the existing file COMMENTS;
                               and open it for storing and
                               retrieving information in
                               records numbered Ø through 2Ø.)
```

After these instructions, you could store a student's response in record number 12 of the file COMMENTS, using instructions such as

```
D: C$(8Ø)                    (Reserve space in memory for
                               storing up to 8Ø characters
                               in string variable C$ .)
C: C$ = %B                   (Assign the contents of the
                               Answer Buffer %B to
                               string variable C$ .)
FO: 12,C$                    (Store the contents of string
                               variable C$ into record 12
                               of the currently open file.)
```

and retrieve such a stored response using this instruction:

```
FI: 12,C$                    (Retrieve the contents of
                               record 12 in the currently
                               open file, and store in
                               string variable C$ .)
```

If you wish to create a new file, deleting any old file by the same name, see the FOX: instruction, discussed earlier in this chapter.

THE MAXIMUM RECORD NUMBER

The maximum record number must be an integer from Ø through 32767, or a simple numeric variable containing any number in the same range. In practice, however, the maximum number of records is usually limited to the number of records available in the specified diskette file. The size of that file was set earlier, when the file was created by a FOX: instruction.

Each time the FIX: instruction re-opens a file it will normally specify the same maximum record number the FOX: instruction used when creating the file. However, PILOT does not force you to be consistent; you may use a different maximum record number each time you open the file, if you wish.

If you create a file specifying one maximum record number, and later re-open the file with FIX: , giving a larger maximum, your storage and retrieval instructions can then specify record numbers corresponding to records which are outside the original file. Any attempt to store information into records beyond the originally created file, and any attempt to retrieve information from such beyond-the-file records, causes an error message (in Author Mode) and raises the Error Flag (in either Author or Lesson Mode).

You may occasionally find it useful to open an existing file of unknown length, using FIX: with an arbitrarily large maximum record number. You can then retrieve information from successive records until the Error Flag is raised to indicate that you have gone beyond the end of the file. In general, however, it is best to create your files with the proper maximum record number in the first place, and then stick to that maximum in future use of the file.

THE STRING VARIABLE

The file name for the existing file to be opened must be stored in a string variable, and only the string variable's name may appear in the FIX: instruction, not the file name itself. A file name must not exceed ten characters. The name must not contain any commas, colons, semicolons, pound-signs (#), dollar-signs, equals-signs, square brackets, asterisks, or spaces. It does not matter whether you type the file name in capital letters, lower-case letters, or both. Internally, all file name letters are always converted to capital letters.

If no file with the specified name is found on the lesson diskette, the FIX: instruction raises the Error Flag. The E conditioner can be used on an instruction following the FIX: instruction to handle this situation.

On a normal, one-disk-drive Lesson Mode system, your lesson's instructions always use files on the same diskette the lesson is stored on. For using systems with additional disk drives, see the appendix Using More Disk Drives.

EXAMPLE

```
D: F$(1Ø)                    (Reserve space to store up to
                               1Ø characters in string
                               variable  F$ .)
D: L$(3Ø)                    (Reserve space to store up to
                               3Ø characters in string
                               variable  L$ .)
T: What is your first name?  (Display this text.)
A: $F$                       (Accept student's response;
                               store in string variable F$ .)
FIX: 9,F$                    (Try to open an existing file
                               whose name is stored in  F$ ,
                               for storing and retrieving
                               information in records Ø-9 .)
JE(1): NEW                   (If Error Flag is up, there is
                               no old file  F$ , so lower
                               Error Flag by evaluating the
                               expression  (1) , and Jump
                               to the label  NEW .)
J: OLD                       (Old file exists, so Jump to
                               section labelled  OLD .)

*NEW                         (Label for new-student section.)
FOX: 9,F$                    (Create and open a new disk file
                               with the name stored in  F$ ,
                               for storing and retrieving
                               information in records Ø-9 .)
T: Well, $F$ , I see this    (Display this text, substituting
: is your first time here.     student's first name for $F$
: What is your last name?      and following space.)
A: $L$                       (Accept student's response;
                               store in string variable L$ .)

FO: Ø,L$                     (Store contents of  L$  into
                               record Ø of the open file.)
J: MAIN                      (Jump to label  MAIN .)

*OLD                         (Label for old-student section.)
FI: Ø,L$                     (Retrieve student's last name
                               from record Ø of the open
                               file; store that information
                               in string variable  L$ .)
T: Well, $F$  $L$ , I see this  (Display this text, substituting
: is not your first time here.    student's first name for  $F$
                                  and last name for  $L$ .)

*MAIN                        (Label for main lesson.)
...                          ...
...                          ... (Main body of lesson.)
```

This example asks the student to type a first name, and then tries to open (using FIX:) an existing file which has the same file name as the student's first name. If there is a file by that name already, the student has used this lesson before, and the lesson then retrieves the student's last name from record Ø of the file. If there is no old file with the student's first name, such a file is created (using FOX:), and the new student's last name is Accepted and stored into record Ø of the file.

Note that only the first ten characters of the student's first name will be placed in string variable F$. This prevents a longer first name from creating an illegally long file name. If you wanted to use longer names in messages to the student, you could Accept the student's first name into two different string variables, one (limited to 1Ø characters) for the file name, and another (Dimensioned to any length) for use in messages:

```
A: $F$ $M$
```

FO: OUTPUT TO OPEN FILE

FO: record number, string variable containing item to be stored

The FO: instruction stores the contents of the specified string variable into the specified record of the diskette file which is currently open. Before the FO: instruction can store an item of information into a file, that file must first be created and opened with a FOX: instruction, or re-opened with a FIX: instruction.

For example, to store the words "This is a recording..." into record 13 of a file that you have already opened, you could use these instructions:

```
D: S$(5Ø)                          (Reserve space to store up to
                                    5Ø characters in string
                                    variable S$ .)
C: S$ = "This is a recording..."   (Store the quoted string in
                                    string variable S$ .)
FO: 13,S$                          (Store the contents of string
                                    variable S$ into record 13
                                    of the currently open file.)
```

In a real lesson, however, you would also have to include instructions to open a file. Suppose you want to store your words into an existing file named MABELLSAYS :

```
D: X$(1Ø)                          (Reserve space to store up to
                                    1Ø characters in string
                                    variable X$ .)
D: S$(5Ø)                          (Reserve space to store up to
                                    5Ø characters in string
                                    variable S$ .)
C: X$ = "MABELLSAYS"               (Store file name MABELLSAYS
                                    in string variable X$ .)
FIX: 2Ø,X$                         (Open the existing file whose
                                    name is stored in X$ , for
                                    storing and retrieving
                                    information in records Ø-2Ø.)
C: S$ = "This is a recording..."   (Store the quoted string in
                                    string variable S$ .)
FO: 13,S$                          (Store the contents of string
                                    variable S$ into record 13
                                    of the currently open file.)
```

THE RECORD NUMBER

The record number must be an integer from Ø through the maximum record number specified by the last file-opening instruction (FIX: or FOX:), or a simple numeric variable containing any number in the same range.

If you create a file specifying one maximum record number, and later open the file again with FIX: , giving a larger maximum, your storage and retrieval instructions can then specify record numbers corresponding to records which are outside the originally created file. Any FO: instruction which tries to store information into records which are beyond the file created by FOX: causes an error message (in Author Mode) and raises the Error Flag (in Author or Lesson Mode).

In general, it is best to create your files with the proper maximum record number in the first place, and then stick to that maximum in future use of the file.

THE STRING VARIABLE

The item of information to be stored in a file record must first be stored in a string variable, and only the string variable's name may appear in the FO: instruction, not the string itself.

Each record can store one string only, up to 255 characters long.

Note: Common PILOT uses the FO: instruction to open a file if no file is currently open, as well as to store information in records of an open file. Apple PILOT requires you to use the special file-opening instructions FOX: and FIX: to open a file; the FO: instruction is used only for storing information into a file record of an already-open file.

EXAMPLE

This rather long example can be used to create a new diskette file and store information in any of its records, or to change the information stored in any record of an existing file.

```
PR: U                              (Convert all response letters
                                    to capital letters.)
D: F$(1Ø)                          (Reserve space to store up to
                                    1Ø characters in string
                                    variable F$ .)
D: X$(255)                         (Reserve space to store up to
                                    255 characters in string
                                    variable X$ .)
*NAME                              (Label for file-name section.)
T: Type the name of your file.     (Display this text.)
A: $F$                             (Accept student's response;
                                    store first 1Ø characters
                                    in string variable F$ .)
FIX: 999,F$                        (Try to re-open existing file
                                    with the name stored in F$ ,
                                    for storing and retrieving
                                    records up to record 999 .)
JE(1): NUMBER                      (If Error Flag is up, no old
                                    file with that name was found,
                                    so lower Error Flag by evalu-
                                    ating the expression (1) and
                                    Jump to label NUMBER .)
```

This example demonstrates most of the concepts you need for storing information in a diskette file. Pay special attention to the many error-handling instructions.

In the section labelled NAME, the lesson Accepts a file name from the student, and tries to open an old file with that name, using a maximum record number large enough to include any existing file. If it succeeds, it lets the student give another file name or choose between erasing the old file and changing records in the old file.

If no old file with the specified name is found, or if the student wants to erase the old file and start a new file with that name, the lesson then Accepts the number of records the new file is to contain. At the end of this section, labelled NUMBER, the lesson creates and opens a new file using the chosen name and the maximum record number calculated from the specified number of files. (Since all files start with record number Ø, the maximum record number is one less than the number of records in the file.)

In the section labelled RECORD, the lesson Accepts a record number from the student, retrieves and displays the current contents of that record, and then Accepts and stores new information in that record. This section repeats until the student presses the RETURN key instead of specifying a record number.

The student-supplied name for the file is automatically limited to ten characters maximum length by assigning the name to a string variable which was Dimensioned to 1Ø characters. The student does not need to know about this limitation, because any retrieval program will also limit the name to ten characters maximum. As long as the student types the same name each time, the file will be found.

See the end of the FI: instruction, next in this chapter, for an example of a lesson which can retrieve the files created by this example.

FI: INPUT FROM OPEN FILE

FI: record number, string variable to receive contents of record

The FI: instruction retrieves the contents of the specified record from the diskette file which is currently open, and stores that information in the specified string variable. Before the FI: instruction can retrieve an item of information from a file, that file must first be created and opened with a FOX: instruction, or re-opened with a FIX: instruction.

For example, to retrieve the information which was earlier stored in record 13 of a file that you have already opened, you could use these instructions:

```
D: S$(5Ø)                (Reserve space to store up to
                          5Ø characters in string
                          variable S$ .)
FI: 13,S$                (Retrieve contents of record 13
                          of currently open file, and
                          store in string variable S$.)
```

If the file whose information you want to retrieve is not already open, your lesson would have to include instructions for opening the correct file. Suppose the information you want is in record 13 of a file named MABELLSAYS :

```
D: X$(1Ø)                (Reserve space to store up to
                          1Ø characters in string
                          variable X$ .)
D: S$(5Ø)                (Reserve space to store up to
                          5Ø characters in string
                          variable S$ .)
C: X$ = "MABELLSAYS"     (Store file name MABELLSAYS
                          in string variable X$ .)
FIX: 2Ø,X$               (Find the existing file whose
                          name is stored in X$, and open
                          it for storing and retrieving
                          information in records Ø-2Ø.)
FI: 13,S$                (Retrieve contents of record 13
                          of currently open file, and
                          store in string variable S$.)
```

THE RECORD NUMBER

The record number must be an integer from Ø through the maximum record number specified by the last file-opening instruction (FIX: or FOX:), or a simple numeric variable containing any number in the same range.

If you create a file specifying one maximum record number, and later open the file again with FIX: , giving a larger maximum, your storage and retrieval instructions can then specify record numbers

corresponding to records which are outside the originally created file. Any FI: instruction that attempts to retrieve the contents of such beyond-the-file records causes an error message (in Author Mode) and raises the Error Flag (in both Author and Lesson Mode).

Sometimes it may be useful to retrieve successive records from a file of unknown length until the Error Flag indicates you have gone beyond the end of the file. In general, though, it is best to create your files with the proper maximum record number in the first place, and then stick to that maximum in future use of the file.

THE STRING VARIABLE

When an item of information is retrieved, it must be stored in a string variable, and that string variable's name must appear in the FI: instruction.

Each record in a file contains a single string, up to 255 characters long. When your FI: instruction retrieves the contents of a record, the string stored in the specified record is then stored in the specified string variable, a process which shortens the record string to the maximum Dimensioned length for the receiving string variable, if necessary.

EXAMPLE

This example lets you retrieve and display the contents of each record in an existing lesson-created file.

```
D: F$(1Ø)                          (Reserve space to store up to
                                     1Ø characters in string
                                     variable F$ .)
D: X$(255)                         (Reserve space to store up to
                                     255 characters in string
                                     variable X$ .)
*NAME                              (Label for file name section.)
T: Type the name of the file       (Display this text.)
:whose contents you want to see.
A: $F$                             (Accept student's response;
                                     assign first 1Ø characters
                                     to string variable F$ .)
E(LEN(%B)=Ø):                      (If only the RETURN key was
                                     pressed, End lesson here.)
FIX: 999,F$                        (Open an existing disk file
                                     with the name stored in F$ ,
                                     for storing and retrieving
                                     records up to record 999.)
TE: Sorry, I can't find that file. (If Error Flag is up, no file
                                     with that name was found, so
                                     display this text.)
JE(1): NAME                        (If Error Flag up, lower flag
                                     by evaluating the expression
                                     (1) and Jump back to NAME .)
C: R = Ø                           (Store initial number Ø in the
                                     record-count R .)
*DISPLA                            (Label for display section.)
FI: R,X$                           (Retrieve contents of record
                                     number R of the file now
                                     open, and store the string
                                     retrieved in string variable
                                     X$ .)
TE: That's all of file $F$ .       (If Error Flag is up, record
                                     R was not in the file, so
                                     display this text.)
WE: 1Ø                             (If the Error Flag is up,
                                     wait 1Ø seconds.)
JE(1): NAME                        (If Error Flag up, lower flag
                                     and Jump back to NAME .)
T: Record #R : $X$                 (Display text, substituting
                                     record number for #R and
                                     one space and record contents
                                     for $X$ and one space.)
C: R = R + 1                       (Increment record-counter R
                                     by 1 .)
J: DISPLA                          (Jump back to DISPLA to
                                     retrieve next record.)
```

This example lesson demonstrates most of the concepts you need for retrieving information which has been stored in a lesson-created diskette file. In the NAME section, the lesson Accepts a file name from the student, and opens an existing file with that name, if the file is on the lesson diskette. The FIX: instruction uses a maximum record number large enough to include all the records of any file. If the student just presses the RETURN key, without typing a file name, that response ends the lesson.

Next, in the section labelled DISPLA, the lesson retrieves and displays each record in the open file, starting from record number Ø . When the Error Flag is raised, this indicates that the lesson has gone beyond the last record in the originally created file.

The student-supplied name for the file is automatically limited to ten characters maximum length by assigning the name to a string variable which was Dimensioned to 1Ø characters. The student does not need to know about this limitation, because the storage program also limited the name to ten characters maximum. As long as the student types the same name each time, the file will be found.

See the end of the FO: instruction, earlier in this chapter, for an example of a lesson which can create or change the files whose contents are retrieved by this example.

CHAPTER 8

EXECUTION TIME COMMANDS

156 GOTO COMMAND
157 Example

159 @ ESCAPE COMMAND
160 Example
161 Example

162 CTRL-C

163 RESET

GOTO COMMAND

The form of the typed response is:

GOTO label or unlabelled destination (any text may follow)

There are several commands available to the student when a lesson is running, in addition to the normal interplay of displayed questions and typed answers. Some of these commands, such as CTRL-C and RESET, can be used at any time, in any lesson. However, the student can use the GOTO command only if the GOTO option has been selected by a PRoblem instruction. The instruction which allows the student to use the GOTO command looks something like this (the instruction may select other options at the same time):

```
PR: G                          (Let the student use the GOTO
                                 execution-time command in
                                 typed responses to Accept
                                 instructions.)
```

For details about setting control options, see the PRoblem instruction, in the chapter Response Instructions.

After the GOTO control option is selected by a PRoblem instruction such as PR:G , the student's typed responses to the Accept instruction are treated in the usual way unless the response begins with the letters GOTO (or goto). A response of the form

GOTO destination

is interpreted as a GOTO command, which causes a Jump to the specified destination. The choices for the destination are:

GOTO label	Jumps to the specified label.
GOTO @A	Re-executes the current Accept instruction.
GOTO @M	Jumps ahead to the next Match instruction.
GOTO @P	Jumps ahead to the next PRoblem instruction.

The command GOTO @A is not useful, since it merely re-executes the same Accept instruction which just accepted the GOTO command. The command GOTO @M Jumps to the next Match instruction, which then treats the GOTO command itself as the accepted response in which to look for the Match words. Any additional text may follow the GOTO command, and will be used as part of that accepted response.

The accepted response, which includes the first characters GOTO and the destination, may be used in the part of the lesson to which execution Jumps. This accepted response is stored in the Answer Buffer (system variable %B) but it is NOT stored in any string variables or numeric variables which may appear in the Accept instruction's object field. The values previously stored in the Accept instruction's object variables are not changed by the Accept instruction which accepts the GOTO command. The Answer Count (system variable %A) is incremented in the usual way by this Accept instruction.

If a destination label is specified, but that label is not found in the lesson, program execution continues at the instruction following the Accept instruction. In this case, the accepted response is still stored only in the Answer Buffer %B , and not in any of the Accept instruction's object variables. If an unlabelled destination is specified, but that destination is not found, the lesson is terminated and the system re-starts, in either Author Mode or Lesson Mode.

The GOTO command can be a very useful tool for developing a lesson. You can use GOTO commands to repeat a section under test again and again, trying different responses to determine what problems may arise. Conversely, you can skip over whole sections of the lesson that have known errors or that you have already tested.

If you select the GOTO option for your own use as a testing aid, you can later eliminate GOTO commands for student use by removing the G specifier from the PRoblem instruction. A student's response beginning with GOTO will then have no special effect, and will be treated like an normal response.

However, the GOTO command can also be a valuable tool for the advanced student. If the lesson tells the students what choices are available, the students can then direct their own progress through the lesson. For example, some students may wish to take a test immediately, to see if they already know the material in this lesson. Others may wish to review certain sections, or skip over sections.

EXAMPLE

```
PR: G U                        (Allow use of the GOTO command
                                 and convert all response
                                 letters to capital letters.)
D: C$(2Ø)                      (Reserve space for storing up
                                 to 2Ø characters in string
                                 variable C$ .)
*MENU                          (Label for menu section.)
T: What part of the lesson     (Display these lines of text.)
: do you want to do now?
T:
T: To choose:                  Just type:
T: This display - - - - - - - GOTO MENU
T: Simple sentences - - - - - GOTO PART1
T: Compound sentences - - - - GOTO PART2
T: Complex sentences- - - - - GOTO PART3
T: Review of lesson - - - - - GOTO REVIEW
T: Test over all material - - GOTO TEST
T: End of the lesson- - - - - GOTO END
T:
T: To stop what you are doing at any time,
: type GOTO MENU in place of a response.
T:
```

```
*CHOICE                          (Label for choice section.)
TH: Please make your choice:     (Display text, leaving cursor
                                   after the colon.)
A: $C$                           (Accept response; and assign
                                   it to string variable C$.)
T: $C$ is not one of the choices.(Bad choice, so display text,
J: CHOICE                          and Jump back to CHOICE .)

*PART1                           (Label for lesson section on
...                                simple sentences.)
...                              . . . . . .
J: MENU                          (Jump back to label MENU to
                                   choose next section.)
*PART2                           (Label for lesson section on
...                                compound sentences.)
...                              . . . . . .
J: MENU                          (Jump back to label MENU to
                                   choose next section.)
...
...
...

*END                             (Label for end of lesson.)
T: You see? Every sentence is not(Display this text.)
: hard labor for life. Good-bye.
E:                               (End of lesson.)
```

Note: the PR: instruction converts response letters to capitals for the convenience of the Match instruction. It is not necessary to do this conversion for the GOTO command to work correctly.

This example was designed to show the use of the GOTO execution-time command. It may not demonstrate the best way to handle choosing from a menu of options, however. For example, you could save the student from typing GOTO with every option by accepting the label name alone, and then using an XI: instruction to Jump to that destination:

```
TH: Please make your choice:
A: $C$
C: D$ = "J:" || C$               (Concatenate the string in
                                   C$ to the string "J:" and
                                   store the result in the
                                   (previously Dimensioned)
                                   string variable D$ .)
XI: D$                           (Execute the contents of D$
                                   as a PILOT instruction.)
```

If the student types a choice of REVIEW , the Compute instruction above makes the string J:REVIEW and stores that string in string variable D$ (whose Dimension instruction you must have added to the beginning of the lesson). The XI: instruction then causes a Jump to label REVIEW .

@ ESCAPE COMMAND

The form of the typed response is:

```
@  (any text may follow)
```

The student can use the Escape command only if the Escape option has been selected by a PRoblem instruction. The instruction which allows the student to use the Escape command looks something like this (the instruction may select other options at the same time):

```
PR: E                            (Let the student use the Escape
                                   execution-time command in
                                   responses to A: instructions.)
```

For details about setting control options, see the PRoblem instruction, in the chapter Response Instructions.

After the Escape control option is selected by a PRoblem instruction such as PR:E , the student's typed responses to the Accept instruction are treated in the usual way unless the response begins with the character @ . A response whose first character is an @ is interpreted as an Escape command, which causes program execution to branch to an author-supplied subroutine labelled SYSX . This branch to the SYSX subroutine, initiated by the Escape command, is equivalent to the following Use-subroutine instruction:

```
U: SYSX                          (Jump to the subroutine
                                   labelled SYSX , and return
                                   to the instruction after
                                   this U: instruction when
                                   the subroutine ends with an
                                   E: instruction.)
```

For more information about using subroutines, read the discussions of the Use and End instructions, in the chapter Control Instructions. Do not confuse the End instruction E: with the PRoblem instruction's Escape option PR: E . The same letter means very different things.

If PRoblem instruction selects the E option, the author must provide a subroutine beginning with the label SYSX . Normally, the subroutine should End with an E: instruction. The E: returns to the instruction after the A: that accepted the @ (Escape) command, unless the E: specifies some other branch destination.

If the Escape command is issued, but no subroutine labelled SYSX is found, program execution continues at the instruction following the Accept instruction which accepted the Escape command. However, the next E: instruction may cause a "return" to the same instruction following this Accept instruction.

The accepted response, which includes the first character @ , may be used in subroutine SYSX . This accepted response is stored in the Answer Buffer (system variable %B) but it is NOT stored in any

string variables or numeric variables which may appear in the Accept instruction's object field. The values previously stored in the Accept instruction's object variables are not changed by the Accept instruction which accepts the Escape command. The Answer Count (system variable %A) is incremented in the usual way by this Accept instruction.

The subroutine SYSX can be as simple as an end-the-program routine, providing you or the student an effective escape. To do that, just put the label SYSX as the last line of the lesson. The SYSX subroutine can also be relatively complex, letting you save and record comments, or pass data to a special function you have written.

EXAMPLE

This example shows a SYSX subroutine which can help you find and correct problems in a lesson you are testing.

```
PR: E U                          (Let the student use the Escape
                                   option, and convert all
                                   response letters to capitals.)
D: B4$(4Ø)                       (Reserve space to store up to
                                   4Ø characters in string
...                                variable B4$ .)
...
...                              (Body of lesson.)
...
*SYSX                            (Label for Escape subroutine;
                                   program jumps here if any
                                   response begins with @ .)
C: B4$ = %B                      (Store contents of Answer Buffer
                                   %B in string variable B4$ .)
C: L = LEN(B4$) - 1              (Set simple numeric variable L
                                   to 1 less than the length of
                                   the string in variable B4$ .)
C: B4$ = B4$(2,L)                (Make new string, using all but
                                   the first character of B4$,
                                   and store resulting string
                                   back into variable B4$ .)
XI: B4$                          (Execute the string in B4$
                                   as a PILOT instruction.)
E: @A                            (End this subroutine by returning
                                   to the same Accept instruction
                                   that initiated this Escape.)
```

This Escape subroutine removes the leading @ sign from the response, and then executes the rest of the response as a PILOT instruction.

In testing your lesson, suppose you discover you forgot to Dimension the variable Q$. After noting the problem, so you will remember to correct it permanently when you are next in the Lesson Text Editor, you can make an immediate temporary correction by typing this response to the next Accept instruction:

```
@D: Q$(3Ø)
```

The leading @ sign causes the program to Escape to subroutine SYSX which removes the leading @ and executes the instruction D:Q$(3Ø) . It then returns you to the same Accept instruction from which you Escaped, so you can continue testing the lesson. You could also use t Escape routine to display the values of variables, Compute new values, or find out other useful information about the lesson, all while the lesson is running.

EXAMPLE

This example shows a SYSX routine that would act as a calculator if the student preceded an expression by the @ sign:

```
PR: EU                           (Let student use the Escape
                                   option; convert response
                                   letters to capitals.)
D: X$(5Ø)                        (Reserve space for string
                                   variable X$ to store
                                   up to 5Ø characters.)
T: You can do a calculation at any (Display all of this
: response point. Just type an       text.)
: "@" followed by any mathematical
: expression (for example, @56/32.9).
: You will see the answer to your
: calculation, and then return to the
: same response point in the lesson.
: Type "@ END" to stop this lesson.
T:
T: If 12 people can write 16 lines of
: corn in 6 hours, how many lines
: can 19 people write in 8 hours?

A:                               (Accept student's response.)
...
...
...
*SYSX                            (Label for Escape subroutine;
                                   program jumps here if any
                                   response begins with @ .)
M: END                           (Did student type END ?)
JY: STOP                         (If Yes, Jump to label STOP .
C: X$ = %B                       (No, so store Answer Buffer %B
                                   in string variable X$ .)
C: L1=LEN(X$)-1                  (Set L1 to one less than the
                                   length of the string in X$ .
C: X$="C: N4= " || X$(2,L1)      (Make a new string, consisting
                                   of "C: N4=" plus all but the
                                   first character of old X$ .
                                   Store new string back into
                                   string variable X$ .)
XI: X$                           (Execute the contents of X$
                                   as a PILOT instruction.)
```

```
TE: That was not a correct          (If Error Flag is up, then
: mathematical expression.            display this text.)
EE(1): @A                           (If Error Flag is up, lower it
                                      by evaluating (1) ; then
                                      return to Accept instruction
                                      which started this Escape.)
T: The answer is #N4                (No error, so display answer.)
E: @A                               (End subroutine  SYSX  by a
                                      return to Accept instruction
                                      which started this Escape.)
*STOP  E:STOP                       (Ends this lesson.)
```

When the student types a response of, say,

```
@ (12*19*8)/(16*6)
```

the program Escapes to the SYSX subroutine, which removes the leading @ sign, makes up the new string "C: N4= (12*19*8)/(16*6)" , and stores that string in variable X$. Then it executes the string stored in X$ as a PILOT Compute instruction, shows the student the result, and returns to the same Accept instruction from which this Escape was initiated.

Note: the PR: instruction converts response letters to capitals for the convenience of the Match instruction. It is not necessary to do this conversion for the Escape command to work correctly.

CTRL-C

The form of the typed response is:

Type C while holding down the CTRL key

If a CTRL-C is typed in the response to any Accept instruction , the lesson currently running is immediately terminated, and the system re-starts itself. To type CTRL-C , type a C while holding down the CTRL key. This execution-time command provides a handy escape from any lesson, while it is running.

You do not need to set any option to enable this command, and you can not set any option to disable this command. It is always available in any lesson, during the response to an Accept instruction. Note that the Accept-Point instruction, AP: , does not accept typed responses, so the CTRL-C command does not work during the response to an AP: instruction.

If you want the student to know about the CTRL-C command, your lesson can use an instruction such as this:

```
T: To stop this lesson at any          (Display this text on
: response, hold down the CTRL key,      the student's screen.)
: and type  C  while continuing
: to hold down the CTRL key.
```


RESET

The response is:

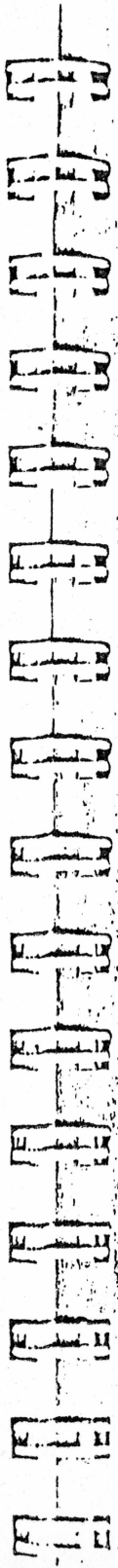
Press the RESET key

When you are running any program in Apple PILOT, you can press the Apple's RESET key any time you want to stop the program. This will cause the system to re-start itself, just as it starts itself when you first turn it on.

The only time you should avoid pressing the RESET key is when the system is busy storing information on the diskette. If the disk drive's red "IN USE" light is off, it is always safe to press the RESET key.

Sometimes a programming error may result in a "loop" which performs some instructions over and over and over. Pressing the RESET key may be the only way to break out of such a loop.

[Note: some Apple keyboards are equipped with a switch which lets you change the normal RESET command to CTRL-RESET (issued by pressing the RESET key while holding down the CTRL key). For the sake of simplicity, the descriptions in this manual assume such keyboards are set to "normal", RESET-only operation.]



CHAPTER 9

HINTS FOR BEGINNERS

166 USING THE STUDENT'S NAME

168 USING NUMBERS

169 COUNTING ANSWERS

171 AN EXAMPLE

This section should prove most useful to those who have little experience with mathematics and computer programming. You will learn three simple operations that make your lessons much more effective. You can use these three operations just as they are, in your own lessons, or you can use what you learn here as a spring-board for understanding many of the more complex operations offered in Apple PILOT.

You need to know about one item before you begin this section: variables. A "variable" is just a fancy term for a place in the Apple where you can temporarily store a number or a word that you may want to use later. The reason such a storage place is called a variable is that you can change or vary what is stored there, while the program is running. You give this storage place a name, usually a one-letter name such as T . If you want to store words in that place, you add a dollar sign to the name, as in the name W$.

Once the storage place has a name, you can put things in it yourself, by using a Compute instruction, or you can let the student put things in it, by using an Accept instruction, as you will soon see. In these instructions, you will tell the Apple to "store this number in T ", or "store this word in W$ ". To use what you have stored, you can put the storage place's name (in this case, T or W$) in a Compute or a Type instruction, and Apple PILOT will automatically substitute the stored number or word wherever that name appears.

Using a storage-place name instead of a number or word can be convenient in two ways. It saves you from typing the actual number or word each place you need it in the lesson, because you can type the one-letter name of its storage place, instead. And you can easily change that number or word throughout a lesson, just by changing what you put in the named storage place.

USING THE STUDENT'S NAME

This is an easy trick which will make your lessons seem much more personal. It lets any of your messages to the student incorporate the student's name. To do this, you will store the student's name in a "string variable", which is the term for a storage place that holds words. We will name this storage place W$.

Before you can use a word-storage place, you must first use a Dimension instruction to tell PILOT how big this place should be, since words come in many different sizes. Names come in many sizes, too, but let's assume that no first name will be longer than, say, 30 characters. This is the instruction which tells PILOT you need a word-storage place named W$, big enough to store words up to 30 characters long:

```
D: W$(30)                    (Reserve a place called W$
                              for storing words up to
                              30 characters long.)
```


The next step is to tell the student to type her or his first name:

```
T: Please type your first name.      (Display this text on the
                                      student's screen.)
```

And now you need to tell the Apple to accept the student's typed name. Here is an instruction which not only accepts the name but also stores it in the place called W$:

```
A: $W$                               (Accept the student's response
                                      and store it in the word-
                                      storage place named W$ .)
```

Notice that you must put an extra dollar sign in front of the normal name of the word-storage place. This is a funny quirk of the Accept instruction, as defined by Common PILOT .

That's all there is to it. From now on, any message in your lesson can use the student's name, which is stored in W$. Here's how you would use the stored name in a Type instruction:

```
T: Well, $W$ , how old are you?      (Display this text on the
                                      student's screen, after
                                      replacing $W$ with the
                                      word stored in the word-
                                      storage place named W$
                                      and one space.)
```

Notice that, once again, you must put an extra dollar sign in front of the normal name for the word-storage place. What's not so obvious, but equally important, is that you must put a space after the place's name, too. In this case, the dollar sign before the name and the space after the name form a special signal to the Type instruction. They say, "Don't display these characters as you usually would, but use these characters as the name of a word-storage place, and display the word stored there, instead." Since the space after the place's name is part of the special signal, that space is not displayed on the student's screen. In the example, the comma will immediately follow the student's name.

In this example, you used the Accept instruction to store a word in your word-storage place. That lets the student type the word which is stored. If you want to put a word in that storage place yourself, from within the lesson, it's even easier. Here's a Compute instruction which stores the word Apple in the word-storage place named W$:

```
C: W$ = "Apple"                      (Store the word Apple in
                                      the word-storage place
                                      named W$ .)
```

Notice that the Compute instruction does not require you to put any extra dollar sign in front of the name of the word-storage place. Several spaces are shown in this Compute instruction, but they are all optional.

You can store new words in a word-storage place as often as you wish during a lesson. Each time you put a new word in that storage place, the old word stored there is forgotten.

USING NUMBERS

Often, you may want your lessons to use a number typed by the student. For instance, the previous instruction asked the student "How old are you?" To use the student's answer, you will store the number part of the answer in a "simple numeric variable", which is the term for a number-storage place. We will name this storage place N .

Since all numbers (as stored by Apple PILOT) are the same size, you do not need to use a Dimension instruction before using a number-storage place. Instead, you can accept the student's age, name a number-storage place, and store the student's age in the number-storage place, all in one instruction:

```
A: #N                    (Accept the student's typed
                          response, and store the
                          response's first number in
                          the number-storage place
                          named N .)
```

Notice that you must put a pound-sign (#) in front of the normal name of the number-storage place. This is another odd quirk of the Accept instruction as defined by Common PILOT. Another thing to notice is that this instruction only recognizes actual numbers, such as 12 or 99, not words that spell numbers, such as twelve or ninety-nine . What gets stored in the number-storage place named N is the first recognizable number typed by the student, ignoring any words or other non-number typing. If the student types

I am twelve years and 2 months and 26 days old.

the number 2 will be stored in the number-storage place named N .

Well, that does it. From now on, any message or calculation in your lesson can use the student's age, which is stored in N . Here's how you would use the stored age in a Type instruction:

```
T: Do you like being #N  years old?     (Display this text on the
                                          student's screen, after
                                          replacing #N with the
                                          number stored in number-
                                          storage place named N .)
```

Here again, you must put a pound sign (#) in front of the normal name for the number-storage place. But again, what's not so obvious is that you must also put an extra space after the place's name. The pound sign before the name and the space after the name are the Type instruction's signal to use these characters as the name of a number-storage place,

and to display the number stored there. Since the first space after the place's name is part of the special signal, that space is not displayed on the student's screen. In the example, there will be only a single space immediately following the student's age.

In this example, you used the Accept instruction to let the student store a number in your number-storage place. But you can as easily put a number in a number-storage place yourself, from within the lesson. Here's a Compute instruction which stores the number 2237.84 in the number-storage place named N :

```
C: N = 2237.84           (Store the number 2237.84 in
                          the number-storage place
                          named N .)
```

Notice that the Compute instruction does not require you to put any extra pound sign in front of the name of the number-storage place. Several spaces are shown in this Compute instruction, but they are all optional.

You can store new numbers in a number-storage place as often as you wish during a lesson. Each time you put a new number in that storage place, the old number stored there is forgotten.

COUNTING ANSWERS

Suppose your lesson had the following instructions:

```
*DOZEN                   (Label for this section.)
T: How many eggs are there
: in a dozen?            (Display this text on the
                          student's screen.)
A:                       (Accept the student's answer.)
M: 12|twelve|TWELVE      (Did the student's answer
                          include any of these three
                          words: 12 , twelve , or
                          TWELVE ?)
TY: That's right!        (If one of the Match words was
                          found in the student's answer,
                          display this text.)
JY: NEXT                 (If one of the Match words was
                          found in the student's answer,
                          Jump ahead to label NEXT, for
                          the next problem.)
T: No, that's not the right
: answer.  Try again.    (If the program gets to here,
                          none of the Match words were
                          in the student's answer, so
                          display this text.)
J: DOZEN                 (Jump back to label DOZEN for
                          another try.)
*NEXT                    (Label for the next section.)
```

If the student does not know how many eggs make a dozen, these instructions will present the question over and over and over, forcing the student to go on guessing indefinitely. Since the idea is to instruct, not to punish, you need a way to limit how many times a student tries to answer the same question.

You can do this very easily, using Apple PILOT's "Answer Count" conditioner on any instruction. The Answer Count conditioner is a number from 1 through 9 that you put immediately after an instruction name. That instruction will then be executed only if the student has tried to answer the same question a number of times equal to the number you wrote after the instruction's name. For example, this instruction:

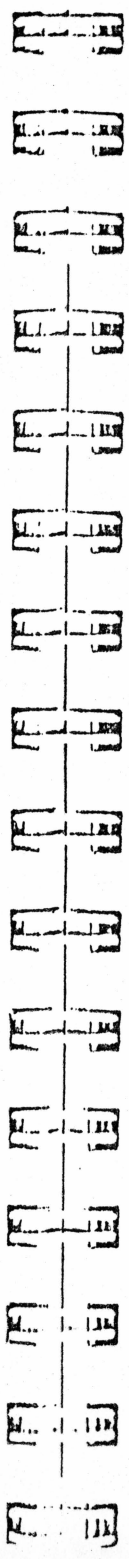
```
T3: No, there are twelve eggs          (Display this text only if the
: in a dozen.                            latest Accept instruction
                                         has been executed 3 times
                                         in a row.)
```

will be skipped over unless the student has just tried to answer the same question for the third time in a row.

So, to rescue the poor student after three unsuccessful guesses, we could add two extra instructions to the previous set:

```
*DOZEN                                 (Label for this section.)
T: How many eggs are there             (Display this text on the
: in a dozen?                            student's screen.)
A:                                     (Accept the student's answer.)
M: 12!twelve!TWELVE                    (Did the student type 12 ,
                                         twelve , or TWELVE ?)
TY: That's right!                      (If Yes, display this text.)
JY: NEXT                               (If Yes, Jump ahead to NEXT .)
T3: No, there are twelve eggs          (If program gets here, answer
: in a dozen.                            was wrong. If this is 3rd
                                         answer, display this text.)
J3: NEXT                               (If 3rd answer, Jump ahead to
                                         the label NEXT .)
T: No, that's not the right            (If program gets here, answer
: answer. Try again.                     was wrong but not 3rd answer
                                         yet, so display this text.)
J: DOZEN                               (Jump back to the label DOZEN
                                         for another try.)

*NEXT                                  (Label for the next section.)
```

Note that a "colon-continuation" of an instruction, as shown in the two-line Type instructions for example, continues all the conditioners and modifiers, as well as the instruction name, in the second line. A continued Type instruction will not appear on the student's screen in the same two-line format. Instead, the displayed words will all go on the first line, if possible, only jumping to the next line if there are still some words that just won't fit on the first line.



AN EXAMPLE

Here's an example of a little lesson portion which stores and uses the student's name, accepts and stores a number from the student, and uses both the name and the number in messages which show the student what two times that number is. It repeats this number-doubling game until the student types the word "end" or until nine numbers have been doubled, and then it goes on to the next section of the lesson. The next section is not supplied in the example. If you try typing this example into the Lesson Text Editor, and running it, it will just end when it goes to the section labelled NEXT .

```
D: F$(3Ø)                              (Create the word-storage place
                                         place named F$ , with space
                                         for storing words up to 3Ø
                                         characters long .)
T: Type your first name, please.       (Display this text on the
                                         student's screen.)
A: $F$                                 (Accept student's typed name
                                         and store it in the word-
                                         storage place named F$ .)
T: Very good, $F$ .                    (Display this text, after
                                         substituting the student's
                                         name (stored in the word-
                                         storage place named F$ )
                                         for $F$ and one following
                                         space.)
*DOUBLE                                (Label for this section.)
T: Type a number and I will double     (Display these lines of text.)
: it, or type "end" to stop.
A: #D                                  (Accept student's typed number
                                         and store it in the number-
                                         storage place named D .)
M: end!END!stop!STOP!quit!QUIT         (Did the student type any of
                                         these six words?)
JY: NEXT                               (If Yes, Jump to label NEXT .)
C: A = 2 * D                           (Make a copy of the number now
                                         stored in the storage place
                                         named D , multiply it by 2 ,
                                         and store the result in the
                                         number-storage place named A.)
T: Okay, $F$ , twice #D is #A .        (Display this text, substitutin
                                         the name stored in word-
                                         storage place F$ for $F$
                                         and one following space, and
                                         substituting the numbers in
                                         number-storage places D and
                                         A for #D and #A and one
                                         space after each.)
```

```
T9: I'm tired of that, $F$ .          (If this is the 9th answer in
: Let's try something else.             a row, display this text
                                        using the name stored in  F$
                                        in place of  $F$  and one
                                        following space.)
J9: NEXT                              (If this is the 9th answer in
                                        a row, Jump to label  NEXT .)
J: DOUBLE                             (Otherwise, Jump back to the
                                        label  DOUBLE , for another
                                        number.)

*NEXT                                 (Label for the next section.)
```

There are a number of limitations to this lesson portion, some of which could be improved by more advanced techniques. For instance, the PRoblem instruction could be used to convert all response letters to small letters, saving you from having to type every Match word in both capital letters and small letters. You could also use the Compute instruction's string-editing feature to make sure the student's name is always displayed the same way, with the first letter capitalized.

More important confusions might arise over the way Apple PILOT displays numbers. Numbers from .1 through 999999 and from -.1 through -999999 , and Ø , are shown normally, after rounding to six digits, if necessary. Thus, a number like 234.56789 is displayed as 234.568 . But if the student types numbers larger or smaller than that, the numbers are converted to "scientific" E-notation. If the student typed 123456789 , for example, the number would be displayed later as 1.23457EØ8 , which means 1.23457 times 1Ø to the eighth power. Students can also use this notation in typing their own numbers, if they wish. This would require some explanation by the teacher, to prepare the student for such numbers.

If the student types more than sixteen digits, any part of the number beyond the sixteenth digit is ignored. For a fancier way to convert a student's response to a number, avoiding this sixteen-digit limitation, see the FLO function, in the chapter Advanced Programming.

If the student types a word with no numbers, or just presses the RETURN key without typing anything, the number stored in the student's number-storage place (named D) is not changed from the number previously stored in it. In this case, the same number typed the previous time is used again. It might be a good idea to put the instruction

```
C: D = Ø                              (Store the number Ø in the
                                        number-storage place
                                        named  D .)
```

in the lesson, just before the Accept instruction which takes the student's number response. That way, if the student typed no number, the number Ø would be used instead of the last number typed. If you want, you can use the Error Conditioner to detect such a no-number response and execute instructions which deal with that situation.

CHAPTER 10

ADVANCED PROGRAMMING

174 CONSTANTS
174 Numeric Constants
174 String Constants

175 VARIABLES
175 Simple Numeric Variables
177 Numeric Array Variables
180 String Variables
182 Substring Variables
184 Pseudo-Variables
185 System Variables
186 %A The Answer Counter
186 %B The Student Answer Buffer
186 %X The Horizontal Coordinate
186 %Y The Vertical Coordinate
186 Using System Variables

188 FUNCTIONS
189 Transcendental Functions
191 Arithmetic Functions
193 String Functions
199 Input Functions

201 OPERATORS
202 Arithmetic Operators
202 Relational Operators
204 Logical Operators
205 String Operator

206 EXPRESSIONS
206 Operator Precedence
208 Limitations on Expressions

The following material requires some knowledge of mathematics and its use in programs. For a less detailed discussion of this material, read the chapters Computation Instructions and Hints for Beginners.

CONSTANTS

NUMERIC CONSTANTS

A numeric constant is simply a number. All numbers in Apple PILOT are "real" numbers; they may be expressed as integers, decimal numbers, or in "scientific" E-type notation (see the examples below).

Examples of numeric constants:

```
13947          (Integer notation.)
43.9237        (Decimal notation.)
7.76E+12       (Scientific notation: this is
                 7.76 times 1Ø to the 12th
                 power, or 7,76Ø,ØØØ,ØØØ,ØØØ.)
```

Example of a numeric constant in use:

```
C: N = 332.786      (Store the number 332.786 in
                      simple numeric variable N .)
```

Apple PILOT's real numbers are limited to these ranges: -3.4Ø282E38 through -1.1755E-38 , Ø , and 1.1755E-38 through 3.4Ø282E38 . Instructions such as Compute and Accept will issue an error message (in Author Mode) and raise the Error Flag (in both Author Mode and Lesson Mode) if they encounter numbers outside these limits.

A maximum of six digits are displayed for a number in integer or decimal notation. A maximum of six digits are displayed for the mantissa (the portion preceding the Exponent) of a number expressed in scientific E-notation. A number with more than six significant digits is rounded to six significant digits BEFORE the number is stored or used. The number is rounded up if the seventh significant digit is 5 or more and down if seventh digit is 4 or less, ignoring any digits beyond the seventh for rounding purposes.

STRING CONSTANTS

A string constant is any sequence of characters, surrounded by double quotation marks ("). Only the characters between the quotation marks are considered to be the string, and not the quotation marks themselves. Some characters cannot be typed into a string directly, but you can use the string functions and operators in an expression to fabricate a string which includes almost any characters.

Examples of string constants:

```
"Apple"          (Character string in quotes.)
"29.732Ø1"       (Any characters, in quotes.)
```

Example of string constant in use:

```
C: S$ = "FAT CHANCE!"      (Store the string FAT CHANCE!
                             in string variable S$ .)
```

VARIABLES

Variables are just places in the Apple where you can store numbers or strings of characters for use later in a lesson. Numeric variables are used for storing numbers, and string variables are used for storing strings of characters. Each variable is named by you. Once you store a number or string in that variable, you can use the variable's name instead of that number or string almost anywhere in your lesson. This saves you a lot of typing, and also lets you change that number or string throughout a lesson by simply changing what is stored in that variable.

Four types of variables may be used with PILOT instructions. These are simple numeric variables, numeric array variables, string variables, and system variables. Each type of variable is discussed in this section.

Every numeric array variable and string variable must first appear in a Dimension instruction to reserve space for it in memory, before it can be used in Compute or in any other instruction.

Your lesson may create a maximum of 5Ø different names for its variables. This includes the total of all simple numeric, numeric array, and string variables, but does not include the individual elements of a numeric array variable or the system variables.

SIMPLE NUMERIC VARIABLES

A simple numeric variable is a named place for storing a single number. The name of the simple numeric variable must be a single letter, or a single letter followed by an integer from Ø through 9 .

When used in the object field of Type or Accept instructions, the name must be immediately preceded by a pound sign (#) and immediately followed by a space or by the end of the instruction line. When the Type instruction is executed, the first space after a simple numeric variable name is not displayed: the stored number is displayed instead of the pound sign, variable name, and following space.

A simple numeric variable is normally created by using a Compute or an Accept instruction to store some number in that variable. For example:

```
C: N = 45                    (Create the simple numeric
                              variable N and store the
                              number 45 in it.)
A: #P                        (Accept student's response;
                              create the simple numeric
                              variable P and store the
                              first response number in it.)
```

If a simple numeric variable is created by using it for the first time in an expression to the right of the assignment indicator (=) in a Compute instruction, or in an Accept instruction which finds no number in the student's response, or in an execution-conditioning expression in any instruction, that variable is created containing some more-or-less random real number between -3E38 and +3E38 .

A simple numeric variable does not need to be Dimensioned, since it stores only one number. However, Dimensioning a numeric array variable (see next section) may influence your use of simple numeric variables. There are 98 number-storage places available for the use of all the simple numeric variables and all the elements of all the numeric array variables created during a lesson. Dimensioning a large numeric array variable may reserve most of the 98 number-storage places, limiting the number of simple numeric variables your lesson can create. It is possible in such a situation to cause a "Too many strings or arrays" error message on creating a simple numeric variable.

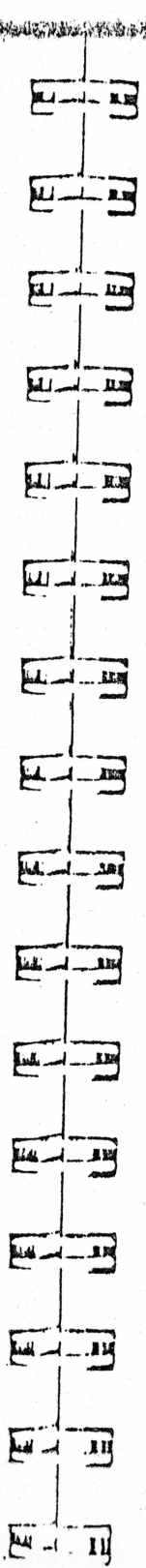
Warning! The simple numeric variable N is the SAME VARIABLE as the numeric array variable element N(Ø) or N(Ø,Ø) . Do not try to use a simple numeric variable and a numeric array variable which share the same name.

Examples of simple numeric variables:

```
N                            (Name is a single letter.)
S9                           (Name is a single letter
                              followed by a single digit.)
```

Example of simple numeric variables in use:

```
C: N = 5                     (Store the number 5 in simple
                              numeric variable N .)
C: Q3 = N + 1                (Get the number stored in simple
                              numeric variable N , add
                              1 to it, and store the sum in
                              simple numeric variable Q3 .)
T: What is #Q3  minus 1 ?    (Display text, substituting
                              the number stored in simple
                              numeric variable Q3 for
                              #Q3 and one following space.)
A: #R                        (Accept student's response, and
                              store first number (if any) in
                              simple numeric variable R .)
```

NUMERIC ARRAY VARIABLES

A numeric array variable is really a set of simple numeric variables sharing the same name. Each of the individual variables (called "elements") in the array is distinguished from the others by a "subscript" number which appears in parentheses after the variable's name. It is sometimes useful to use a numeric array variable in your lesson, because the lesson can choose among the different elements of the array just by changing a number.

Numeric array variables may be one-dimensional (requiring a single number subscript) or two-dimensional (requiring a subscript consisting of two numbers separated by a comma). A one-dimensional numeric array variable might be used to store a list of numbers, such as the score on each problem in a lesson. A two-dimensional numeric array variable mig be used to store a table of numbers, such as the closing stock prices for IBM on each day of the week.

The name of the numeric array variable must be a single letter, or a single letter followed by an integer from Ø through 9 . The name of an individual element in an array is the array variable's name followed by a subscript specifying that element.

The subscript for an element of a one-dimensional array is a single number in parentheses. The subscript for an element of a two-dimension array is two numbers separated by a comma, with one set of parentheses around the entire subscript. A number in a subscript may be replaced b a simple numeric variable name, but not by an expression or by an element of a numeric array variable. This feature allows you to choose an array element by simply changing the subscript number which is store in a simple numeric variable.

Examples of numeric array variable elements:

```
W6(23)                       (The 24th element of the one-
                              dimensional numeric array
                              variable W6 . Remember the
                              first element is W(Ø) .)
Q(N)                         (The number stored in simple
                              numeric variable N is used
                              as the subscript denoting an
                              element of one-dimensional
                              numeric array variable Q .)
A(2,5)                       (The 6th element in the 3rd row
                              of the two-dimensional array
                              variable A . First element
                              in row Ø is A(Ø,Ø) , then
                              A(Ø,1) , A(Ø,2) , etc. The
                              elements of row 1 are A(1,Ø)
                              A(1,1) , A(1,2) , etc.)
```

Warning! The first element of a numeric array variable, specified by subscript (Ø) or (Ø,Ø) , is the SAME VARIABLE as the simple numeric variable by the same name. Thus N and N(Ø) and N(Ø,Ø) are the same variable.

Before an array variable can be used by Compute or any other instruction, it must first appear in a Dimension instruction to reserve enough number-storage places for its elements. The Dimension instruction specifies the maximum subscript number for each of the array's dimensions. The minimum subscript number for each array dimension is Ø . The maximum subscript number is limited by the amount of number-storage space available. There are 98 number-storage places to be used by all the simple numeric variables and all the elements of all the numeric array variables in a lesson. A numeric array whose maximum subscript is Dimensioned as (9,8) has 9Ø elements (9+1 * 8+1), leaving only eight number-storage places for all the other numeric variables in the lesson.

The Dimension instruction creates all the elements of the numeric array variable, but it does not store any number in those elements. Until you use a Compute instruction to store a number in an element of the array, that element will contain some more-or-less random real number from -3E38 to +3E38 .

The results of these instructions:

```
D: A(6)                    (Reserve space for storing up to
                             7 numbers in one-dimensional
                             numeric array variable  A .)
C: A(1) = 32.34            (Store 32.34 in element A(1) .)
C: A(2) = 557              (Store 557 in element A(2) .)
C: A(5) = 8.921            (Store 8.921 in element A(5) .)
```

might be pictured like this:

A (Ø)	(1)	(2)	(3)	(4)	(5)	(6)
	32.34	557			8.921	

Each "empty" array element will actually contain some random number until you store a number in that element.

The results of these instructions:

```
D: B(3,2)                  (Reserve space for storing up to
                             12 numbers in two-dimensional
                             numeric array variable  B .)
C: B(Ø,2) = 43             (Store 43 in element B(Ø,2) .)
C: B(1,Ø) = 7.8            (Store 7.8 in element B(1,Ø) .)
C: B(2,3) = 9219           (Store 9219 in element B(2,3) .)
```

might be pictured like this:

B	(Ø,Ø)	(Ø,1)	(Ø,2)	(Ø,3)
			43	
	(1,Ø)	(1,1)	(1,2)	(1,3)
	7.8			
	(2,Ø)	(2,1)	(2,2)	(2,3)
				9219

Again, each "empty" array element will actually be filled with some random number until you store your own number there.

Numeric array variable elements cannot appear in the object field of Accept or Type instructions. Accept must first store a number in a simple numeric variable; then Compute can store that number in an element of the numeric array variable. Compute can also store the contents of a numeric array variable element into a simple numeric variable which can then be used in a Type instruction.

Example of a numeric array variable in use:

```
D: P(3Ø)                   (Reserve space for 31 elements
                             of numeric array variable  P .
                             The elements go from  P(Ø)
                             through  P(3Ø) .)
...                         ...
...                         ...
*PROB5                     (Label for test problem 5.)
TH: 23 + 13 =              (Display this text, leaving the
                             cursor after the equals sign.)
A: ØG                      (Accept student's response, and
                             store first number in simple
                             numeric variable  G .)
T(G<>36): No, try again.   (If student's answer now stored
                             in  G  is not equal to 36,
                             display this text.)
JC: PROB5                  (If answer  G  is not 36 (same
                             condition), Jump back to the
                             label  PROB5 .)
C: P(5) = %A               (Right answer, so store Answer
                             Count %A in the array element
                             denoted by subscript 5 .)
*PROB6                     (Label for test problem 6.)
```

When the lesson above is finished, each element of numeric array variable P will contain the Answer Count for the corresponding test

problem. The Answer Count tells how many attempts the student made before giving the correct answer.

To display the scores stored in numeric array variable P , you might use instructions such as these:

```
C: N = Ø                          (Store first subscript Ø in
                                    simple numeric variable N .)
*SCORE                            (Label for scores section.)
C: X = P(N)                       (Get score stored in element
                                    N of numeric array variable
                                    P , and store it in simple
                                    numeric variable X .)
T: Score on problem #N was #X     (Display text, substituting
                                    the number stored in N for
                                    #N , and the number in X
                                    for #X .)
C: N = N + 1                      (Add 1 to the number in N ,
                                    and store it back in N .)
J(N<31): SCORE                    (If the number in N is less
                                    than 31, Jump back to SCORE.)
```

Notice that we had to first store the number from numeric array variable element P(N) in simple numeric variable X before we could display that score in the Type instruction.

STRING VARIABLES

A string variable is a string-storage place where you can store a single string of characters for use in your lesson. It is often helpful to use a string variable's name in your lesson, in place of the string that is stored in that variable. This lets you change that word or phrase throughout the lesson, simply by changing the string which is stored in that named variable.

The name of the string variable must be a single letter, or a letter and a single integer from Ø through 9, and must be immediately followed by a dollar ($) sign. You may use a numeric variable and a string variable with the same basic name in a lesson; the string variable is always distinguishable by the dollar sign after its name.

Examples of string variables:

```
A$                                (Name is a single letter,
                                    followed by a dollar sign.)
S8$                               (Name is a single letter,
                                    followed by a single digit,
                                    followed by $ .)
```

When used in the object field of Type or Accept instructions, the string variable's name must be immediately preceded by a dollar sign ($) and immediately followed by a space or by the end of the instruction line. Thus, the string variable name S$ appears as S . When the Type instruction is executed, it replaces the preceding dollar sign, the string variable name, and the first space after the name by the string of characters currently stored in that string variable. When the Accept instruction is used to store a student's response in a string variable, the response receives full Accept editing unless all editing has been suspended by Accept's eXact modifier (AX: S).

Before a string variable can be used by Compute, Type, or any other instruction, it must first appear in a Dimension instruction to reserve space for its characters in memory. The Dimension instruction specifies the maximum number of characters that may be stored in the variable. This maximum must be a number from 1 through 255 characters. The Dimension instruction creates the string variable, but it does not store any characters in the variable. Until you use an Accept or Compute instruction to store a string of characters in a string variable, it contains no characters at all.

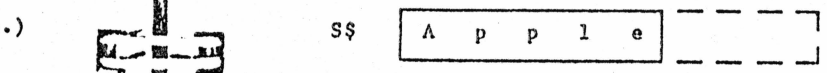
There are a total of 6ØØ character-storage locations for all the characters stored by all the string variables in a lesson. When you Dimension a string, the specified maximum number of character locations are made permanently unavailable for use by any other string variable. However, Apple PILOT first temporarily stores each string in the portion of the 6ØØ locations which is NOT reserved, before it can be stored in the locations reserved for a string variable. This means there must at all times be enough UNRESERVED locations to store the largest string you will want to put in a string variable.

A string is stored in a string variable from left to right (the first character or leftmost character of the string is stored in the string variable's first position). A length-of-string indicator is stored with the string to tell PILOT how long the stored string is. This length is reset each time a new string is stored in the variable. If the string being stored is longer than the maximum length Dimensioned for the string variable, the extra (rightmost) characters of the string are discarded.

The result of these instructions:

```
D: S$(8)                          (Reserve space for storing
                                    strings up to 8 characters
                                    long in string variable S$.)
C: S$ = "Apple"                   (Store the string Apple in
                                    string variable S$ .)
```

might be pictured like this:



(Current length of S$ is 5 characters.)

Example of string variables in use:

```
D: A$(2Ø)                    (Reserve space for storing
                               strings up to 2Ø characters
                               long in string variable  A$ .)
D: B$(5Ø)                    (Reserve space for storing
                               strings up to 5Ø characters
                               long in string variable  B$ .)
C: A$ = "apples"             (Store the string  apples
                               in string variable  A$ .)
C: A$ = A$ !! " and pears"   (Make a copy of the string in
                               string variable  A$ ; join
                               the string  and pears  to it.
                               Store the resulting string
                               back into  A$ , replacing
                               A$'s previous contents.)
T: What are your             (Display these lines of text.)
: favorite fruits?
A:  $B$                      (Accept student's response and
                               store in string variable B$ .)
T:  Yes, $B$ are nice.       (Display text, replacing $B$
                               and the first following space
                               with the string now in
                               string variable B$ .
T:  Have you ever tried $A$ ?  (Display text,
                               substituting the contents of A$
                               where $A$ and space appear.)
```

In the example above, if the student replies that "mangos and papayas" are favorite fruits, the Apple will respond:

```
Yes, mangos and papayas are nice.
Have you ever tried apples and pears?
```

And if the student replies, "I think fruit is ghastly!" , the Apple will respond:

```
Yes, I think fruit is ghastly! are nice.
Have you ever tried apples and pears?
```

Substring Variables

A string variable may sometimes be treated like a one-dimensional array of characters. When used this way, the string variable can be thought of as a family of "character-variables", each storing one character.

The result of these instructions:

```
D: S$(8)                     (Reserve space for storing
                               strings up to 8 characters
                               long in string variable S$.)
C: S$ = "Apple"              (Store the string  Apple  in
                               string variable  S$ .)
```


might be pictured like this:

S$ (1) (2) (3) (4) (5)

A	p	p	l	e

(Current length of S$ is 5 characters.)

The complete set of these character-variables is the string variable itself, of course, and smaller sets can form "substring variables". For convenience, character-variables and substring variables often both go under the name substring variables.

Individual character-variables within the string variable share the string variable's name, immediately followed by a subscript consisting of a single number in parentheses. The subscript number corresponds to a character position in the stored string, a number from 1 (denoting the first character) through the number of characters currently stored in the string variable. Thus, the character-variable storing the first character in string variable S$ is S$(1) :

S$(1) (1)

A

A substring variable shares the string variable's name, immediately followed by a subscript consisting of two numbers separated by a comma, with one set of parentheses around the whole two-number subscript. The first number gives the string position of the substring's first character-variable, and the second subscript gives the number of character-variables in the substring variable. The value of either subscript number can be from 1 through the number of characters currently stored in the main string variable, but the sum of the subscript numbers must also not exceed by more than 1 the number of characters currently stored. Thus, the substring variable storing three characters, starting with the second character in string variable S$, is S$(2,3) . This substring variable also includes character-variables S$(2) through S$(4) :

S$(2,3) (2) (3) (4)

p	p	l

Any subscript number can be replaced by a simple numeric variable, but not by a numeric array variable element or by an expression.

Examples of character-variables and substring variables:

```
R$(2)                    (Specifies the second character
                          of the string stored in
                          string variable  R$ .)
X$(3,12)                 (Specifies a substring, twelve
                          characters long, beginning
                          with the 3rd character stored
                          in string variable  X$ .)
```

Example of substring variables in use:

```
D: Q$(2Ø)                (Reserve space for storing
                          strings up to 2Ø characters
                          long in string variable  Q$ .)
D: Z3$(2Ø)               (Reserve space for storing
                          strings up to 2Ø characters
                          long in string variable  Z3$ .)
D: L5$(2Ø)               (Reserve space for storing
                          strings up to 2Ø characters
                          long in string variable  L5$ .)
C: Q$ = "LABORATORY"     (Store the string  LABORATORY
                          in string variable  Q$ .)
C: Z3$ = Q$(1,5)         (Make a copy of the 5-character
                          substring  LABOR  starting at
                          character 1 of the string in
                          Q$ , and store it in string
                          variable  Z3$ .)
C: L5$ = Q$(4,7)         (Make a copy of the 7-character
                          substring  ORATORY  starting
                          at character 4 of the string
                          in  Q$ , and store it in
                          string variable  L5$ .)
T: This is a $Q$ .       (Display this text, replacing
: Let's have more $Z3$    $Q$ , $Z3$ , and  $L5$  with
: and less $L5$ .         the strings now in string
                          variables Q$, Z3$, and L5$ .)
```

The above example should display the following message on the screen:

```
This is a LABORATORY.
Let's have more LABOR
and less ORATORY.
```

Pseudo-Variables

A pseudo-variable is a character-variable or substring variable (see the previous section) which appears on the lefthand side of the assignment indicator (=) in a Compute statement. This allows you to change a single character or group of characters in a stored string, without affecting the other characters in that string.

If the new string of characters being stored in the substring variable is longer than the substring variable, the extra (rightmost) characters of the new string are discarded. If the new string is shorter than the substring variable, spaces are added to the right end of the new string until it is as long as the substring variable.

Note that you cannot CREATE strings or characters by the use of a pseudo-variable; you can only CHANGE characters which are already currently stored in the string of which the pseudo-variable is a portion. If a pseudo-variable includes any character positions which do not currently have characters stored in them, an error condition results.

For examples of pseudo-variables, see the previous section's examples of substring variables and character-variables.

Example of pseudo-variables in use:

```
D: PØ$(2Ø)               (Reserve space for storing
                          strings up to 2Ø characters
                          long in string variable PØ$.
C: PØ$ = "COMPUTER"      (Store the string  COMPUTER
                          in string variable  PØ$ .)
C: PØ$(3,4) = "workhorse" (Replace 4-character substring
                          starting at character 3 of
                          string variable  PØ$  with
                          string constant  workhorse
                          after discarding all but the
                          4 leftmost characters.)
T: $PØ$                  (Display the string now stored
                          in string variable PØ$ .)
```

The above example, when run, should display the following word:

```
COworkER
```

If the third instruction had been

```
C: PØ$(3,4) = "rk"
```

the example would have displayed this, instead:

```
COrk  ER
```

SYSTEM VARIABLES

In addition to all the variables that you can create yourself (see the previous sections), Apple PILOT maintains four of its own variables for internal use and for use in your instructions. These four variables are called "system variables". System variables do not have the usual, author-created variable names, and they cannot be used in all of the same ways a normal variable can be used. Here are the four system variables:

%A The Answer Counter

The numeric value stored in the %A system variable is the Answer Count: the number of successive times the most recent Accept instruction has been executed without a different Accept instruction being executed. The Answer Count records how many times in a row a student has responded to the same question. Before your lesson has executed its first Accept instruction, the value stored in %A is Ø . Thereafter, the value the system stores in %A is always an integer that is 1 or greater. The value of %A for the first nine responses to any Accept instruction can be automatically tested as a condition for executing other instructions, by using the Answer Count conditioner.

%B The Student Answer Buffer

Before your lesson executes its first Accept instruction, there are no characters stored in the %B system variable. Thereafter, the string stored in %B by Apple PILOT is the student's most recent typed response, as received and edited by an Accept instruction. The next Match instruction searches the contents of the Answer Buffer for any occurrence of the author-supplied Match text. The Accept instruction will store up to 8Ø characters (before editing) in %B .

%X The Horizontal Coordinate

Before your lesson executes its first Accept Point (AP:) instruction, the numeric value stored in the %X system variable is Ø . Thereafter, the number stored in %X is the Horizontal or X Coordinate supplied by game controller #Ø at the time of the most recent Accept Point instruction. The Accept Point instruction stores in %X the graphics x-coordinate of the cross-hair which appears on the screen, controlled by the game paddles, a number from Ø through 56Ø .

%Y The Vertical Coordinate

Before your lesson executes its first Accept Point (AP:) instruction, the numeric value stored in the %Y system variable is Ø . Thereafter, the number stored in the %Y system variable is the Vertical or Y Coordinate supplied by game controller #1 at the time of the most recent Accept Point instruction. The Accept Point instruction stores in %Y the graphics y-coordinate of the cross-hair which appears on the screen, controlled by the game paddles, a number from Ø through 511 .

Using System Variables

A system variable can be used in expressions, but the contents of a system variable should only be changed by the Apple PILOT system. This means that you should never place a system variable on the lefthand side of an assignment operator (=) in a Compute instruction, and you must not use a system variable in the object field of an Accept instruction. A system variable may not appear with subscripts, so you must not specify substrings of the Answer Buffer %B . The value stored in a system variable can, of course, be assigned to any appropriate author-created variable, using a Compute instruction, and that assigned value can then be used in all the usual ways.

While it is not recommended, it is possible to use Compute to store a string in system variable %B . Your lesson might edit a response, for instance, and then store the edited response back into %B . It is also possible for Compute to store a number in system variable %A . An Escape (SYSX) subroutine, for example, might reduce the Answer Count by 1 so the Escape response itself would not be counted as an attempted answer. A number stored in %A is truncated (any decimal portion is discarded), and numbers greater than 32767.9 or less than -32767.9 are stored as 1 . Since the Answer Count conditioner must be a single digit, it will not work properly with %A values greater than 9 or less than 1 . Again, this use of the system variables is discouraged.

System variables cannot be used directly in a Type instruction to place the stored value into the displayed text. If you want a system variable's value to appear in a Type instruction, you must have previously assigned the system variable's value to a standard author-created variable, in a Compute instruction. Assign the values of %A , %X , and %Y to numeric variables (e.g., C: Z= %A). Use a string variable for %B (e.g., C:Y$ = %B). Then place your author-created variable in the Type instruction. Creating your own variables with the names A , B$, X , and Y will not interfere with the system variables having similar names.

Example of system variable %A in use:

```
C: R = RND(1ØØ)                  (Pick a random number from Ø
                                  through 99, and store it in
                                  simple numeric variable R .)
*START                           (Label for this section.)
T: Guess my number, Ø through 99.  (Display this text.)
A: #G                            (Accept student's response
                                  and store it in G.)
TE: Type a number, please.       (If Error Flag is up, no number
                                  was typed, so display text.)
JE(1): @A                        (If Error Flag up, lower it by
                                  evaluating the expression (1)
                                  and Jump back to last A: .)
J(G=R): RIGHT                    (If guess G is equal to right
                                  answer R , Jump to RIGHT .)
T(G>R): No, that's too large.    (If guess G is greater than the
                                  answer R , show this text.)
T(G<R): No, that's too small.    (If guess G is less than right
                                  answer R , show this text.)
T(%A<2Ø): Try again.             (If the number stored Answer
                                  Counter %A is less than 2Ø,
                                  display this text.)
JC: START                        (If Answer Count is less than 2
                                  Jump back to label START .)
T: Too bad.  My number was #R .  (This is 2Øth wrong answer, so
                                  give up and display this text
                                  substituting right answer R
                                  for #R .)
J: NEXT                          (Jump to section labelled NEXT.
```

```
*RIGHT                          (Label for right section.)
C: S = 21 - %A                  (Right answer, so Compute score,
                                  based on number of guesses.
                                  Store score in simple numeric
                                  variable S .)
C: T = %A                       (Store Answer Count %A in
                                  simple numeric variable T .)
T: You guessed it in #T tries,  (Display this text, substituting
: so you get #S points.           Answer Count (now stored in T)
                                  for #T and score stored in
                                  S for #S .)
*NEXT                           (Label for next section.)
```

FUNCTIONS

Apple PILOT will do a variety of special operations for you, each selected by typing a three-letter name. These special operations are called "functions". Apple PILOT functions can calculate the cosine of a given angle, generate a random number, convert a number into a string, find a number within a string, note the setting of a game control, and perform many other useful tasks.

Each function needs certain items of information in order to do its operation for you. These items are called the function's "arguments". From one to three arguments may follow the function's name, the entire set of arguments surrounded by a pair of parentheses. Multiple arguments are separated by commas.

For example, to calculate the sine of an angle of 1.57 radians, you could use the SIN function in a Compute instruction such as this:

```
C: X = SIN(1.57)                (Calculate the sine of a 1.57
                                  radian angle, and store the
                                  result in simple numeric
                                  variable X .)
```

You can also use this function in place of a number in many other expressions and instructions. Here are some sample instructions:

```
C: X = 5 * 12/SIN(1.57) +16     (Evaluate the expression on the
                                  right of the equals sign, and
                                  store the result in X .)
G(X > SIN(1.57)): DX,Y          (If X is greater than the
                                  sine of a 1.57 radian angle,
                                  Draw a line to x=X,y=Y .)
J(Y < 5 * 12/SIN(1.57) + 16): PART2  (If Y is less than the result
                                  of the remaining expression,
                                  Jump to the label PART2 .)
T(SIN(X)<.75): Try a larger angle.   (If the sine of X radians is
                                  less than .75, display text.)
S(SIN(X)): 3Ø,1ØØ               (If the sine of X is non-
                                  zero, play this note.)
```

Note: if the argument of a function is a single number or variable name, the parentheses around the argument may be omitted. This is not recommended, but it might occasionally be useful if you need to reduce the number of waiting operations in an expression. Functions have the highest priority among operations, so that SIN1.5**2 is executed as (SIN(1.5))**2 . See the section on Expressions for more details about precedence among operations.

Each function requires specific kinds of arguments. A numeric argument (shown in the following descriptions as X) may be a number, a simple numeric variable, an element of a numeric array variable, or an expression which yields a number. A string argument (shown in the following descriptions as X$) may be a quoted string, a string variable, a sub-string variable, or an expression which yields a string. Expressions are discussed in the section following the descriptions of the functions.

When a function completes its operation, either a number or a string is the result. The function is said to "return" this number or string. For instance, the function SIN(X) returns a number which is the sine of the angle X .

In the following discussion, a "valid real number" is any whole number, decimal number, or number expressed in scientific notation, whose value is in the approximate range from 3.4E38 to 1.2E-38, or is zero, or is in the approximate range from -1.2E-38 to -3.4E38 . Those are the approximate limits on numbers which can be Accepted into a numeric variable without causing an Arithmetic Error. Certain calculations or functions may occasionally yield valid numbers slightly exceeding those limits.

Most specific limits are given as the numbers would be stored or used: after rounding to six significant digits. Thus, if an upper limit is stated to be 123456 , any number less than 123456.5 (the number 123456.499, for instance) will be rounded to 123456 and be within the limit.

TRANSCENDENTAL FUNCTIONS

These Apple PILOT functions measure angles in radians, not in degrees. An angle of pi radians is equal to an angle of 18Ø degrees. If you want to convert an angle of D degrees into its equivalent radian measure R , use this conversion:

```
R(radians) = D(degrees) * (3.14159/18Ø)
```

These are only a few of the standard trigonometric functions. Consult any book on trigonometry for the ways to calculate other quantities using these functions. For example, the tangent of an angle can be found by dividing the angle's sine by its cosine.

SIN(X) Returns a number in the range from -1 through +1 which is the sine of the angle X . The single numeric argument X is the size of an angle, expressed in radians. The value of X can be any valid real number. If X exceeds the range -1Ø2942 through 1Ø2942 , SIN(X) returns the value Ø. Example:

```
C: S = SIN(Ø.317)          (Compute the sine of an angle
                            of Ø.317 radians, and store
                            in numeric variable S .)
```

COS(X) Returns a number in the range from -1 through +1 which is the cosine of the angle X . The single numeric argument X is the size of an angle, expressed in radians. The value of X can be any valid real number. If X exceeds the range -1Ø294Ø through 1Ø294Ø , COS(X) returns the value Ø. Example:

```
T(COS(A)=.249): Good shot!  (If the cosine of A radians is
                             .249, display this text.)
```

ATN(X) Returns a number in the range from -.785398 through 1.57Ø8 which is the radian measure of the angle whose tangent is X . This number is also called the arctangent of X . The single numeric argument X is the tangent of an angle. The value of X can be any valid real number.

A tangent does not uniquely determine a corresponding angle (in general, any two angles which differ by pi radians have the same tangent). The ATN(X) function usually returns the smallest positive angle having a tangent of X . However, if X is in the range from Ø through -1 , ATN(X) returns the smallest negative angle whose tangent is X . If X exceeds the range -5.32511E18 through 5.32511E18 , ATN(X) returns the value Ø. Example:

```
C: A = ATN(T)              (Compute the angle, in radians,
                            whose tangent is T, and store
                            in numeric variable A .)
```


ARITHMETIC FUNCTIONS

SGN(X) Returns -1 if X is less than zero, returns Ø if X equals zero, or returns 1 if X is greater than zero. The value of X can be any valid real number. Example:

```
C: B = A * SGN(R)          (If R is greater than Ø then
                            B = A ; if R equals Ø then
                            B = Ø ; if R is less than Ø
                            then B = -A .)
```

ABS(X) Returns a positive number which is the absolute value of X If X is zero or positive, the absolute value of X is just X . If X is negative, the absolute value of X is -X . The value of X can be any valid real number. Example:

```
U(ABS(A)>36Ø): ERROR       (If A is greater than 36Ø or
                            less than -36Ø, Use the
                            subroutine labelled ERROR .)
```

FIX(X) Returns an integer in the range from 32767 through -32767, found by "truncating" X . Truncation is performed by discarding any part of X to the right of the decimal point, without rounding. Thus, FIX(5.933) returns the value 5 , FIX(-3.957) returns the value -3 . The value of X can be any valid real number in the range -32767.9 through 32767.9 . Remember that the value of X is rounded to six significant digits BEFORE being used. Example:

```
C: I = FIX(N)              (Truncate the value of N and
                            store in variable I .)
```

INT(X) Returns an integer in the range from 32767 through -32767 which is the largest integer less than X or equal to X Thus, INT(7.946) returns the value 7 , and INT(-4.12) returns the value -5 . The value of X can be any valid real number in the range -32766.9 through 32767.9 . Example:

```
C: A = INT(L)              (Find the largest integer less
                            than or equal to L, and stor
                            in numeric variable A .)
```

Note: at present, there is an error in this function. If X (after rounding to six significant digits) is an INTEGER less than -255 , INT(X) returns X-1 instead of the correct value X . Thus, FIX(-256) returns the value -257, and an X value of -32767 (after rounding) causes an Arithmetic error. This problem does not affect the usual, non-integer values of X .

RND(X) Returns a random number picked from 32768 positive numbers in a range determined by the value of X . The value of X can be any valid real number less than 32768.Ø . For any X less than 1 , RND(X) returns a random real number in the range from Ø through Ø.999969 . For any X equal to 1 or greater than 1 , RND(X) returns an integer from Ø through FIX(X)-1 . Example:

```
C: X = RND(54Ø) + 2Ø        (Pick a random integer from Ø
                             through 539, add 2Ø to it,
                             and store in variable X .)
```

SQR(X) Returns the square root of X (the number which, when multiplied by itself, yields X). The value of X can be Ø or any valid real number greater than Ø . This limits the values returned by SQR(X) from Ø through about 2E19 . Example (finds the length of the hypotenuse H for a triangle with sides of lengths A and B):

```
C: H = SQR((A*A)+(B*B))     (Compute A*A and B*B, add them
                             together, take the square
                             root of that sum, and store
                             in variable H .)
```

LOG(X) Returns a number between about 39 and -39 which is the base 1Ø logarithm of X (the power to which 1Ø must be raised to yield X). The value of X can be any valid real number greater than Ø .

```
C: I = LOG(V/R)             (Divide the value of V by the
                             the value of R , compute the
                             logarithm of the quotient,
                             and store in variable I .)
```

LN(X) Returns a number between about 89 and -88 which is the natural logarithm of X (the power to which e must be raised to yield X). e is approximately 2.71828 . The value of X can be any valid real number greater than Ø .

```
C: F = 2.3 + LN(14.29)      (Compute the natural logarithm of
                             14.29, add 2.3, and store the
                             sum in numeric variable F .)
```

EXP(X) Returns a number in the range from Ø to about 6.8E38 which is e raised to the power X . e is approximately 2.71828 . The value of X can be any valid real number. If X is greater than 89.4159 , EXP(X) returns Ø . Example:

```
C: N = EXP(Y) * 5           (Raise e to the power Y ,
                             multiply by 5, and store in
                             simple numeric variable N
```

STRING FUNCTIONS

ASC(X$) Returns a number in the range from Ø through 255 which is the decimal ASCII code for the first character of X$. The characters normally typed from the keyboard have decimal ASCII code numbers ranging from 32 through 126. See the appendix ASCII Character Codes for the code number that corresponds with each character. Example:

```
J(ASC(%B)=78): NOPE         (If the decimal ASCII code of
                             the first character in the
                             Answer Buffer %B is
                             78, the student has typed a
                             capital N, so Jump to the
                             section labelled NOPE .)
```

CHR(X) Returns the character whose decimal ASCII code number is FIX(X) . See the appendix ASCII Character Codes for the character that corresponds with each code number. The value of X can be any valid real number in the range -32767.9 through 32767.9 , but all of the available characters can be obtained using values of X in the range Ø through 127 . For values of X in the range 128 through 255, CHR(X) returns characters identical to those which would be returned by CHR(X-128), except that they are stored with the higher ASCII code.

If X is greater than 255, the remainder left after dividing FIX(X) by 256 is used in place of X , so CHR(256) returns the character whose ASCII code is Ø, CHR(257) returns the character whose ASCII code is 1, and so on. If X is less than Ø, the remainder left after dividing FIX(X) by -256 is subtracted from 256, and that number is used in place of X , so CHR(-1) returns the character whose ASCII code is 255, CHR(-2) returns the character whose ASCII code is 254, and so on.

The characters corresponding to ASCII codes less than 32, when sent to the screen, do not appear as visible characters. Instead, many of them act as screen-handling instructions to clear the screen, erase a line, do a line feed or carriage-return, etc. You cannot type these characters from the keyboard, but you can place them in

strings and in string variables by using the CHR function. Example:

```
D: S$(1Ø)                    (Reserve space to store up to
                              1Ø characters in string
                              variable  S$ .)
C: S$ = CHR(8)               (Store the character whose
                              ASCII code is 8 (a backspace
                              character) in string
                              variable  S$ .)
C: S$ = S$ !! S$             (Concatenate two s$ strings
                              and store back in string
                              variable S$ .  S$  now contains
                              two back-space characters.)
T: The U.S.A. consists                (Display text.)
TH: of     states.$S$ $S$ $S$ $S$ $S$  (Display text,
                                        then do double
                              backspace 5 times and leave
                              cursor there.  This places
                              the cursor in the blank area
                              between "of" and "states".)
A: #N                        (Accept student's response;
                              assign number to variable N.)
T(N=5Ø): Very good.          (If N is 5Ø, display text.)
```

Note: there is currently an error in the CHR function. If you concatenate anything to a CHR(X) which forms the first character of a string, the CHR(X) character is changed by the character which follows it. In an instruction such as C:X$=CHR(32)!!"APPLE" , the resulting string is AAPPLE . This problem only affects a CHR(X) which is the FIRST character in a string being formed by concatenation. This situation can be avoided by first storing the CHR(X) in a string variable, and then using that string variable's name in the concatenation instead of CHR(X) :

```
C: S$ = CHR(32)              (Store the character whose
                              ASCII code is 32 (a space)
                              in string variable  S$ .)
C: X$ = S$ !! "APPLE"        (Concatenate the string  APPLE
                              to the string now in  S$ ,
                              and store the resulting
                              string in  X$ .)
```

FLO(X$) Returns any valid real number which is the value of the first number found in X$. Starting at the beginning of the string, FLO(X$) examines each character until it finds the first numeric digit. It then examines the preceding character to see if it is a decimal point or minus sign. If a decimal point precedes the numeric digit, the character before that is examined to see if it is a minus sign. FLO(X$) then forms a number using all the characters from its current position up to the first subsequent character which cannot be interpreted as part of that number. Thus, FLO("My 3rd child is 14 years old.") returns the value 3 .

If no numeric digit is found in X$, FLO(X$) causes an error message (in Author Mode) and the Error Flag is raised (in both Author Mode and Lesson Mode).

Periods and hyphens in the string may be interpreted as parts of a number if they occur in the character just before the number. FLO("This is no.4 on the list.") returns the number .4 . FLO("too many--12, in fact--are missing") returns the number -12 . A comma is not recognized as part of a number, so that FLO("I've got $23,244,Ø56 in the bank.") returns the value 23 . The Compute instruction's string-editing option can be used remove commas from a string.

Since the string is scanned from left to right, collecti and evaluating digits along the way, any number of leadi zeros cause no trouble. However, if the number collecte at any point along the string exceeds the limits for re numbers, the instruction is terminated (with an error message, in Author Mode), even though a subsequent exponent, say, would have brought the number back to within limits. This restricts you to about 38 or 39 digits preceding the decimal point, starting at the firs non-zero digit, and restricts you to about 37 or 38 zero before reaching the first non-zero digit, starting at th decimal point. Thus a 1 followed by 39 zeros, and a decimal point followed by 39 zeros, both cause an error, regardless of what comes next.

Note that C:N=FLO(XB) does not put the same number in as does A:#N . The Accept instruction scans only the first 16 numeric characters, including the sign, decima point, and any leading zeros, and ignores any part of th number beyond the sixteenth numeric character.

At most two minus signs can appear in a number, one as the first character, and another after an E as the sign of an exponent. FLO(X$) tolerates any number of decimal points in a number, ignoring all except the rightmost decimal point. FLO("1..2.3.4") returns the value 123.4 A decimal point after the E in an exponent ends the number. Only the first two digits following an E are us in the exponent; any subsequent digits are ignored. Th FLO("1E2345") returns the value 1E23 . Either capital

or lower-case e is interpreted as a numeric exponent character. Example:

```
C: N = FLO(S$!!"E12")          (Concatenate the string "E12"
                                 to the end of the string in
                                 string variable S$, find the
                                 first number in that combined
                                 string, and store the number
                                 in numeric variable N .)

T(FLO(%B)<27): Too small.      (If the first number in the
                                 Answer Buffer %B is less
                                 than 27, display this text.)
```

STR(X) Returns a string, from 1 to 12 characters in length, which represents the number X as it would normally be displayed on the screen. The value of X can be any valid real number. The value of X is always changed to standard Apple PILOT format before STR(X) converts the number into a string, even if X is an actual number (not a numeric variable). Thus, STR(123456789) returns the string 1.23457EØ8

You can use the STR function any time you want to place a number directly into a string. In particular, a number must always be converted to a string before you can store it in a diskette file. Example:

```
C: S3$ = STR(%A)               (Convert the Answer Count %A
                                 to a string, and store in
                                 string variable S3$ .)
FO: 5,S3$                      (Store the contents of S3$ in
                                 record 5 of the diskette file
                                 which is currently open.)
```

Conversely, in order to do any arithmetic operation on a number in a string, you must first convert that part of the string to a number, using the FLO function. Example:

```
FI: 5,R4$                      (Retrieve the contents of
                                 record 5 from the disk file
                                 which is currently open.)
C: N = FLO(R4$) * 12           (Find the first number in R4$ ,
                                 multiply by 12, and store in
                                 simple numeric variable N .)
C:T6$ ="Score:" !! STR(N)      (Convert number N to a string,
                                 concatenate to "Score:", and
                                 store in string variable T6$.)
FO: 5,T6$                      (Store the contents of T6$ in
                                 record 5 of the diskette file
                                 which is currently open.)
```

The two Compute instructions above could have been combined:

```
C: T6$ = "Score:" !! STR( FLO(R4$) * 12)
```


INS(N,T$,P$) Starting at character number FIX(N), looks in target string T$ for the first sequence of characters that matches pattern string P$. Returns the value Ø if no such sequence is found, or returns an integer from 1 through LEN(T$) indicating the position of the character within T$ where the matching sequence begins.

The integer returned by the INS function refers to character positions within T$ numbered from the first character in the string, which is position number 1, and NOT from position N . The value of N merely indicates the character position within T$ from which t begin the search for P$. If N is 1 , the search begins at the first character of T$. If N is 14 , th search begins at character number 14 , ignoring the first thirteen characters of T$. In either case, if the INS function finds the pattern string P$ starting at th twenty-third character of T$, it returns the value 23 . The search always proceeds toward the right through T$.

The value of N can be any valid real number less than 32768.Ø . If N is less than 1 , the value 1 is used instead. If N specifies a starting position beyond the last character position in T$, the INS function returns the value Ø . Unlike most numeric arguments, starting position N must NOT be an expression. It may be a number, a simple numeric variable, or an element of a numeric array variable.

Unlike most string arguments, target string T$ may not be an expression (INS returns the value Ø if T$ is an expression). It may be a quoted string, a string variabl or a string variable sub-string. There are no limitatio on the pattern string P$. If pattern string P$ is longer than the target string T$, the INS function returns the value Ø .

You can use the INS function to determine the existence of a given word within any string. Since Apple PILOT treats the value Ø as False, and any non-zero value as True, the number returned by INS can be used as a simple true/false expression modifying other commands. Example

```
T(INS(1,%B,"TIGER")): Right.   (If the word TIGER is foun
                                 in the Answer Buffer %B
                                 the INS function returns
                                 non-zero number, so disp
                                 the text.)
```

That use of INS is similar to the Match instruction, but INS gives you extra information about where the pattern was found in the response. The number returned by INS

can also be used to change a certain word found in the target string. Example:

```
D: S$(3Ø)                    (Reserve space in memory to
                               store up to 3Ø characters in
                               string variable S$ .)
D: R$(5)                     (Reserve space to store up to
                               5 characters in R$ .)
C: S$ = "A hungry wolf in sheep's clothing."  (Store in S$.)
C: R$ = "wolf"                                (Store in R$.)
C: P = INS(1,S$,R$)          (Find character position where
                               pattern string wolf starts
                               in target string S$ ; store
                               that position in variable P .)
C: S$(P,4) = "lamb"          (Replace 4 characters of S$
                               starting at position P with
                               the 4 characters lamb .)
T: $S$                       (Display the string S$ .)
```

The INS function should have returned the value 1Ø, the position in S$ where the first occurrence of the sequence of characters wolf started. We then used that number 1Ø as the starting position of a sub-string in S$ to be replaced by the string lamb . The new word had to be the same length as the word it was replacing, to avoid changing any other characters in S$.

LEN(X$) Returns an integer which is the number of characters currently stored in X$. This number may be as small as Ø , for the null string "" , and may be as large as the maximum size specified for X$ when space was reserved for that string in a Dimension instruction (cannot exceed 255). The LEN function helps you deal with strings or string-portions of unknown length, or whose length changes from one running of a lesson to the next. Example:

```
T(LEN(XB)>7): Try a shorter word.   (If the number of
                                      characters in Answer
                                      Buffer XB is less
                                      than 7, display text.)

C: L5 = LEN(T$) - 4                 (Subtract 4 from the number of
                                      characters now in string
                                      variable T$ , and store the
                                      result in variable L5 .)
T(INS(L5,T$,"Zaarb")):Good. (If the word Zaarb is found
                              in the last 5 characters of
                              string T$ , display text.)
```


INPUT FUNCTIONS

PDL(X) Returns an integer from Ø through 255 which represents the position of game control number X . The value of X can be any valid real number in the range -32767.4 through 32767.4 , but actual game controls are numbered Ø through 3 . The value of X is rounded (not truncated) to the nearest integer value. If X is greater than 3 , the value 3 is used instead. If X is less than Ø , the value Ø is used instead. If no game control number X is connected to your Apple, PDL(X) always returns the maximum value 255 .

The game controls are sometimes called "paddles" (hence the name of the PDL function) because they are used to control the position of game paddles in certain computer games. Any Ø-15ØK ohm variable resistance can be used as a "paddle", if properly connected to the Apple. See the Apple II Reference Manual for electrical details. If your Apple is equipped with the usual single pair of game controls, those controls are numbered Ø and 1 .

Note that, for graphics purposes, you may wish to use the Accept Point (AP:) instruction to read the position of the game controls. However, the PDL function lets you read a game-control position without putting cross-hairs on the student's screen. Example:

```
T(PDL(Ø)<14Ø): Higher!      (If game control Ø is set at
                              a position represented by
                              a number less than 14Ø,
                              display text.)
C: X = PDL(Ø) * 2.2          (Multiply the position of game
                              control Ø by a graphics
                              scaling factor of 2.2, and
                              store in variable X .)
C: Y = PDL(1) * .75          (Multiply the position of game
                              control 1 by .75, and store
                              in variable Y .)
G: C3;PX,Y                   (Set plotting Color to White!
                              and Plot a dot in that color
                              at the point x=X,y=Y .)
```

BTN(X) Returns the number 1 if the button on game control number X is being pressed, or returns Ø if that button is not being pressed. The value of X can be any valid real number in the range -32767.4 through 32767.4 , but only the numbers Ø through 2 can specify actual game controls which may have buttons. The value of X is rounded (not truncated) to the nearest integer value. If X is greater than 2 , the value 2 is used instead. If X is less than Ø , the value Ø is used instead. If no game control number X is connected to your Apple, BTN(X) always returns the "pressed" value 1 .

Note that, of the four game controls, only the first three have corresponding buttons. Of course, button number 1 does not actually have to be mounted on game control number 1. Each of the numbered buttons is simply any switch properly connected to the Apple. See the Apple II Reference Manual for electrical details. If your Apple is equipped with the usual single pair of game controls, those controls are numbered Ø and 1 .

You can use the BTN function to let the student indicate any simple two-state response. For instance, the student might press the game-control button to indicate when the game-control "paddle" or knob is at the desired position. Example:

```
*WAIT  J(BTN(1)=Ø): WAIT      (If button on game control #1
                                is not being pressed, Jump
                                back to the label  WAIT .)
```

KEY(X)

Returns the number 1 if any keyboard key has been pressed since the last Accept or Wait instruction ended, or returns Ø if no key has been pressed. This function responds to any key except CTRL, SHIFT, or RESET. The value of X can be any valid real number. It does not matter what value of X you specify; this pseudo-argument is not used by the function in any way.

Once a key has been pressed, KEY(X) will continue to return the value 1 until an Accept, Accept Point, or Wait instruction resets the keyboard. KEY(X) will return Ø after these instructions (until the next keypress), whether they were terminated by a response or by waiting longer than the maximum response time set by a PR:Tn instruction. Example:

```
T:To go on, press any key.  (Display text.)
*HALT  J(KEY(9)=Ø): HALT    (If no key has been pressed
                              since the last Accept or
                              Wait, Jump to label HALT.)
```

Note that the example above will wait for a keypress only if no key has been pressed since the execution of the last Accept or Wait instruction.

TIM(X)

Returns a number which is the time, in seconds, that elapsed while the last Accept or Accept Point instruction waited for the student to complete a response. The value of X can be any valid real number. It does not matter what value of X you specify; this pseudo-argument is not used by the function in any way.

After an Accept instruction, TIM(X) returns the number of seconds which elapsed before the student pressed the RETURN key to terminate the response. The minimum time returned is Ø.125 seconds. If the response was terminated by waiting longer than the maximum response time set by a PR:Tn instruction, TIM(X) returns the value Ø . Example:

```
PR:T5 U                       (Set maximum response time to
                               5 seconds, convert response
                               letters to capital letters.)
T:What color is an orange?    (Display text.)
A:                            (Accept student's response.)
C: N = TIM(1)                 (Assign response time to
                               simple numeric variable N.)
T(N=Ø): Your time is up.      (If student did not complete
                               response in 5 seconds,
                               display this text.)
M: ORANGE                     (Did student type ORANGE ?)
TY:You got the right answer   (If Yes, display this text.)
TN:You got the wrong answer   (If No, display this text.)
T(N):in only #N  seconds      (If N is not Ø, display text,
                               substituting the response
                               time  N  for  #N .)
```

After an Accept Point instruction, TIM(X) returns the number of seconds which elapsed before the student pressed either of the two game-control buttons to terminate the response. The minimum time returned is Ø.Ø4 seconds. If the response was terminated by waiting longer than the maximum response time set by a PR:Tn instruction, TIM(X) returns the value Ø .

OPERATORS

Many calculations, comparisons, and other arithmetic and string operations involving constants, variables, and functions are indicated in your lessons by "operators". An operator is indicated by either on or two special symbols. Three operators (unary minus sign, unary plus sign, and logical NOT) affect only the element which follows the operator. All the other operators are placed between the two elements which are used in carrying out that operation.

Each operator requires an element or two elements of a certain type. numeric element (indicated in these discussions as X or Y) may be numeric constant, a simple numeric variable, an element of a numeric array variable, or a function or expression which yields a numeric result. A string element (indicated in these discussions as X$ or Y$ may be a string constant, a string variable, a sub-string variable, or function or expression which yields a string result. Expressions are discussed in the section following this one.

ARITHMETIC OPERATORS

The arithmetic operators handle all the usual "calculator" operations: addition, subtraction, multiplication, division, and exponentiation. Two additional operators indicate the sign of an element.

These operators must be used with numeric elements. The value of an element may be any valid real number, and the result of the operation must also be a valid real number. This means that the elements and the operation results should be in the range from about -1.18E-38 through about -3.4E38, or be Ø , or be in the range from about 1.18E-38 through about 3.4E38 .

Operator Symbol	Name of Operation	Form of Operation	Example in Compute Instr.	(Amount Assigned to Variable P5)
+	unary plus sign	+X	C: P5 = +13.3	(Positive 13.3)
-	unary minus sign	-X	C: P5 = -L4	(Negative L4)
+	addition	X + Y	C: P5 = T1 + 7	(T1 plus 7)
-	subtraction	X - Y	C: P5 = 19 - Q	(19 minus Q)
*	multiplication	X * Y	C: P5 = I * R	(I times R)
/	division	X / Y	C: P5 = V / P	(V divided by P)
**	exponentiation	X ** Y	C: P5 = B ** 8	(B raised to the power 8)

The unary minus-sign multiplies the element after the sign by -1 . The unary plus-sign is rarely used, since it does not change the element after it in any way.

If the exponentiation operation (X ** Y) would result in a value greater than about 6.8E38 or less than about -6.8E38, the result is Ø . Also, the intermediate result X*LN(Y) must not exceed the range for valid real numbers. If Y is zero, the result is one, except that if X is zero the result is always zero. If X is negative, then Y must be an integer in the range -32767 through 32767 . When X is negative, the result is positive for even Y , and negative for odd Y .

Note: there is currently an error in the exponentiation routine which yields no result at all for a certain very small range of values that would normally yield a result near 1.18E-38 or near -1.18E-38

RELATIONAL OPERATORS

The relational operators let you compare two elements to see if they are the same, or to see which element would appear first in an ordered list. You can compare two numeric elements, or you can compare two string elements, but you must not compare two elements of different types.

A relational operator is used to make an "assertion", such as "X=Y". The operation then compares X and Y to determine whether or not the assertion was true. If the assertion was true, the operation yields the number 1 as the result. If the assertion was false, the number Ø is the result.

Two numeric elements may be compared, and the operation is fairly obvious. The numeric value of each element is determined, and those two values are compared. Thus the assertions 9=9 and 9>3 (9 is greater than 3) are both true, so the result in both cases would be 1 .

Two string elements are compared in the following way. Each element is evaluated and reduced to a single resulting string. Then, the ASCII code number for the first character in one string is compared to the ASCII code number for the other string's first character. This is a normal, numeric comparison, just like comparing two numeric elements. If the first characters have different code numbers, the comparison ends there. If those numbers are the same, the second character in each string is compared in the same way. If the two strings are of different lengths, the shorter string has spaces (ASCII code 32) added to the end of it for the purposes of comparison. Thus, the assertions "Apple "="Apple" and "Apple">"Apple"!!CHR(29) are both true, and the result would be 1 in both cases. See the appendix ASCII Character Codes for the code number corresponding to each character.

Operator Symbol	Name of Comparison	Form of Comparison	Example in Compute Instr.	(Amount assigned to Variable P5)
=	is equal to	X = Y X$ = Y$	C: P5 = 5.2=5.2 C: P5 = "AB"="AC"	(True, 1) (False, Ø)
<>	is not equal to	X <> Y X$ <> Y$	C: P5 = N<>4 C: P5 = "A"<>E$	(1 unless N=4) (1 unless E$(1) is letter A)
<	is less than	X < Y X$ < Y$	C: P5 = 5.2<5.1 C: P5 = "Z"<"a"	(False, Ø) (True, 1)
>	is greater than	X > Y X$ > Y$	C: P5 = 5.2>5.1 C: P5 = "99">"AA"	(True, 1) (False, Ø)
<=	is less than or equal to	X <= Y X$ <= Y$	C: P5 = 5<=5 C: P5 = "$"<="&"	(True, 1) (True, 1)
>=	is greater than or equal to	X >= Y X$ >= Y$	C: P5 = 17>=17.Ø1 C: P5 = S$>="A"	(False, Ø) (1 if ASC(S$(1) is 65 or more)

Note that, in a Compute instruction such as C:Q=5 , the equals sign is NOT the relational "is equal to" operator but the assignment or "is assigned the following value" indicator. Almost every Compute instruction consists of the instruction name, followed by a colon, followed by the name of a variable, followed by the assignment indicator (=), followed by the numeric or string element to be evaluated and assigned to or stored in the named variable.

The element to be evaluated may be an assertion containing a relational operator such as the "is equal to" (=) operator. Thus, in the Compute instruction C:Q=X=Y , the first = is the assignment indicator, while

XPRESSIONS

pressions are combinations of constants, variables, functions, and erators which perform arithmetic calculations or string manipulations, even both. Of course, you have been seeing expressions all through ls manual: every function argument is an expression, everything signed to a variable is an expression, and every parenthesized ndition for an instruction's execution is an expression. An operator d its related elements form an expression which, when evaluated, may used as an element in a larger expression.

PERATOR PRECEDENCE

expressions such as 3+4*2 , you don't know whether the result is or 11 unless you know the order in which multiple operators are ecuted. This order among operators in an expression is called recedence". The precedence of each operation in an expression can be termined in three ways:

1. Before anything is evaluated, each user variable and system variable is replaced by the value assigned to it. Each function is also replaced by the value which that function returns (this may require first evaluating any expression that appears as an argument to the function).

2. All operations within a set of parentheses (like this) are carried out before proceeding to the rest of an expression. Thus, the result of (3+4)*2 is 14, while the result of 3+(4*2) is 11. If you want to make the order of execution absolutely clear in an expression, use parentheses.

If parenthesized sub-expressions are "nested" one inside the other, the sub-expression in the innermost set of parentheses is evaluated first. Then the result of that sub-expression is used as an element in evaluating the surrounding expression, and so on. Thus, the expression 7*(6/(5-2)) becomes 7*(6/3) after the first step, then becomes 7*2 after the next step, and the final result is 14.

3. When the order of execution in an expression or sub-expression is not set by parentheses, the operations are carried out in the following sequence:

Execution Order	Operator Symbols	Name of Operation
First	^ NOT	logical NOT
	-	unary minus sign
	+	unary plus sign
Second	**	exponentiation
Third	*	multiplication
	/	division
Fourth	+	addition
	-	subtraction
	||	concatenation
Fifth	=	: equal
	<>	: not equal
	<	relational : less than
	>	comparisons: greater than
	<=	: less than or equal
	>=	: greater than or equal
Sixth	& AND	logical AND
	| OR	logical OR

When operations from different precedence levels appear in the same expression, with no parentheses to alter the precedence, the operation with the highest precedence is executed first. This is the operation which appears closest to the level marked "First" in the preceding table. Then the operation with the next highest precedence is executed.

In the expression 3-7&3**3*2>7 first the exponentiation 3**3 is carried out. The resulting expression is 3-7&27*2>7 . Then the multiplication 27*2 is done, making the expression 3-7&54>7 . Next the subtraction 3-7 is carried out, leaving 4&54>7 . Then the comparison 54>7 is executed, and the expression is 4&1 . Finally, the logical AND gets its chance: the resulting, completely evaluated expression is 1 .

You can see that several of the six levels of precedence contain more than one operator. Operations of equal precedence are simply executed starting with the leftmost operation first, and proceeding toward the right.

In the expression 3*4+6/6*2/4-8 , first all the multiplications and divisions are carried out, working from left to right. 3*4 yields 12. Then 6/6 yields 1, which result is multiplied by 2 to give 2, and that result is divided by 4 to give .5 . This leaves the expression 12+.5-8 . Finally, all the additions and subtractions are executed, and the resultant expression is 4.5 .

LIMITATIONS ON EXPRESSIONS

In general, an operation in an expression is carried out as soon as it becomes clear the operation does not depend on the outcome of another operation. In the expression 3+4*5-6/7**8+1 , the first operation (addition) must wait to see if its second element is followed by an operation of higher precedence. Then, the second operation (multiplication) must also wait, to see if its second element is followed by an operation of still higher precedence. We now have two operations waiting to be executed. The third operation (subtraction) is not of higher precedence than either of the waiting operations, so both of the waiting operations can be carried out immediately.

In Apple PILOT, there can be a maximum of nine operations waiting to be executed. "Operations" which count in this context are operators, parenthesized subexpressions, and functions, if their execution must await the outcome of another operation. For example, the expression 1+2*3-4/5+3*4-6/7+5*6-8/9+4*3**2+1 causes no problems, because there are never more than a couple of operations waiting. But the expression 1+(1+(1+(1+(1+2*3)))) will not be executed, because the multiplication operation is the tenth waiting operation. (The waiting operations are +, (, +, (, +, (, +, (, +, and *.) In an expression such as 1+SIN(X+2*3) , the function name and the left parenthesis together add two waiting operations.

Note: if the rules of expression syntax are not followed, the expression may be evaluated anyway, with unpredictable results.

APPENDIX A

ASCII CHARACTER CODES

210 SCREEN COMMAND CHARACTERS

213 SCREEN MODES

215 PRINTING CHARACTERS

veral operations in Apple PILOT use the number codes by which the
ple recognizes characters. They follow a standard coding scheme for
aracters, called the ASCII character codes. For example, the ASC
nction in PILOT returns the standard ASCII decimal code (the code can
so be expressed in binary, hexadecimal, etc.) for the first character
ored in a specified string variable. Similarly, the CHR function
turns the character which corresponds to a specified ASCII decimal
de number.

CREEN COMMAND CHARACTERS

he "characters" corresponding to the ASCII decimal code numbers from Ø
hrough 31 are not printing characters which are displayed on your
creen, but commands to the display device. These tell the Apple to
"clear the screen", or "set a text viewport", etc. The command is carried
out when a string containing the command character is sent to the screen
or display, by the Type instruction. You can also send one of these
commands to the screen by using the Graphics instruction's X command,
followed by the appropriate code number.

Most of the commands affect the text cursor only, and operate within the
text viewport that is currently set. Since the text cursor itself is not
visible, you will not usually see the effect of a command until the Type
instruction or a response typed on the keyboard causes a printing
character to be displayed on the screen at the new cursor position.

The text cursor is also moved by any displayed Type text or keyboard
response text, and by any Graphics command that moves the graphics
cursor when the text viewport is set to occupy the full screen. However,
note that the text cursor does not join the graphics cursor after a
graphics move until the end of the G: instruction which contains the
graphics move. The text cursor does NOT join the graphics cursor after
Graphics instructions which contain no graphics moves. The only
exception to this is the Graphics X3Ø,x,y command, which moves the
text cursor to location x,y immediately.

The text displayed by the Graphics instruction's T command is not
influenced by these screen commands, even though they affect the display
of all other text.

Decimal Code	Standard Name	Command Name	Effect on the Screen
Ø	NUL		(No action.)
1	SOH		(No action.)
2	STX		(No action.)
3	ETX		(No action.)
4	EOT		(No action.)
5	ENQ	Return + linefeed	Move the text cursor to the left edge of the text viewport, and one line down.
6	ACK		(No action.)
7	BEL	"Bell"	Make a "Beep" sound on the Apple's built in speaker.
8	BS	Backspace	Move the text cursor one character position to the left.
9	HT		(No action.)
1Ø	LF	Linefeed	Move the text cursor one line down, without moving to the right or left.
11	VT	Clear to end of screen	Clear the text line to the right of the text cursor, and clear all of the text viewport below that line.
12	FF	Clear screen	Clear the entire text viewport and move the text cursor to the top left corner of the text viewport.
13	CR	Return	Move the text cursor to the left edge of the text viewport, and one line down.
14	SO	Normal video	Set normal print (white characters on black background) for Type text and keybonrd response text.
15	SI		(No action.)
16	DLE	Indent	Use the ASCII code of the first character that follows, minus 32, as the number of spaces to insert starting at the current text cursor position.
17	DC1		(No action.)
18	DC2	Inverse video	Set inverse print (black characters on white background) for Type text and keyboard response text.
19	DC3		(No action.)
2Ø	DC4	Release viewport	Set the text viewport to occupy the full screen. Releases the viewport set by CHR(22) and CHR(23). To release the Viewport set by G:V1,r,t,b use G:V , not CHR(2Ø) .

Decimal Code	Standard Name	Command Name	Effect on the Screen
21	NAK		(No action.)
22	SYN	Set top left of viewport	Set the top left corner of the text viewport to the text cursor. Does not affect the bottom or right edges of the text viewport. Unlike G:Vl,r,t,b CHR(22) does not tell the outer PILOT system about the set viewport. This viewport should be cleared using CHR(2Ø) before returning to the main lesson.
23	ETB	Set bottom right of viewport	Set the bottom right corner of the text viewport to the text cursor. Does not affect the top or left edges of the text viewport. Unlike G:Vl,r,t,b CHR(23) does not tell the outer PILOT system about the set viewport. This viewport should be cleared using CHR(2Ø) before returning to the main lesson.
24	CAN		(No action.)
25	EM	Home	Move the text cursor to the top left corner of the text viewport.
26	SUB		(No action.)
27	ESC	Screen mode	If the first character that follows is 1 , 2 , or 3 , set the screen mode to that number for displaying Type text or keyboard response text: Mode 1: Normal display. Mode 2: Overprint. Mode 3: Exclusive Or. These are described after character 31. Ignore (do not print) the first character that follows if it is not 1 , 2 , or 3 .
28	FS	Forward space	Move the text cursor one character position to the right.
29	GS	Clear to end of line	Clear the text line to the right of the current text cursor position.
3Ø	RS	Goto XY	Use the ASCII code of the first character that follows, minus 32, as a horizontal character position relative to the left edge of the text viewport. Use the ASCII code of second character that follows, minus 32, as a line number relative to the top edge of the text viewport. Move the text cursor to that position or the edge of the viewport closest to that position. The top line is Ø , and the first character position on a line is Ø . After a Graphics command such as X3Ø,32,38 the text cursor is moved immediately, not waiting for the end of the Graphics instruction.
31	US	Reverse linefeed	Move the text cursor one line up, without moving to the right or left.

SCREEN MODES

Screen Mode 1 is the default mode normally used by the system. In addition, Mode 1 is the only mode used for text displayed by the Graphics instruction's T or X commands. To change the screen mode used for displaying Type instruction object text and response text, you can use either a Graphics instruction or a Type instruction. The Type instruction must be preceded by a Compute instruction which stores CHR(27) in a string variable.

To set Screen Mode	Use this Graphics instruction	Or use this Type instruction after executing D:M$(1) C:M$=CHR(27)
1	G:X27,49	T:M 1
2	G:X27,5Ø	T:M 2
3	G:X27,51	T:M 3

Mode	Effect on the Screen
1	Normal display: each character, with its own small background, completely replaces whatever was on the screen i that place. This is the mode that is always used by the Graphics instruction's T command.

Mode	Effect on the Screen
2	Overprint display: shows each character's "on" dots in addition to any screen dots that were already "on" in that place. In normal video, the dots of the character itself are "on", while the background dots are "off". In reverse video, the opposite is true.

Because of Apple's method of producing color, this mode does not work for printing characters on colored backgrounds. A white character on any background other than black is practically invisible. However, you can use this mode for underlining a word, say, by Typing the word, then back-spacing, and then Typing the underline character in Mode 2. You can also create many other interesting "combination" characters by overprinting two or more characters in the same place.

This mode is usually bad for displaying keyboard response text, since it moves a white square cursor (all dots "on") in front of the response characters. In normal Mode 1, the old cursor is replaced by the typed response character, but in Mode 2 the typed character is just printed "on top of" the old, all-white cursor.

The main use of Mode 2 is for displaying text in a part of the screen crossed by a few, thin graphics lines. In Mode 1, the background of each character may erase gaps in the lines. In Mode 2, the lines will cross right through the characters without problems.

3	Exclusive OR display: shows each character's "on" dots in addition to any screen dots that were already "on" in that place, except turns "off" any screen dot specified as "on" by BOTH the former screen image and the character.

This mode is useful for printing text across a background consisting of large areas of both black and white. The characters will show up as white when printing over a black background, and as black when printing over a white background. This mode is not usually very effective for displaying keyboard response text.

PRINTING CHARACTERS

The characters corresponding to the ASCII decimal code numbers from 3 through 127 are the standard printing characters which can be displaye on the screen. By knowing the ASCII code numbers for these character you can use the CHR function to put them in strings which let you display all the characters which are not directly typeable on the Appl keyboard.

Decimal Code	Standard Name	Decimal Code	Standard Name	Decimal Code	Stand Name
32	SPACE	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	DEL

Note: the DEL character, ASCII code 127, is displayed as an inverse SPACE, appearing as a white square on the screen.

The ASCII code numbers from 128 through 255 correspond to screen comma and printing characters which repeat the set from 0 through 127.

APPENDIX B

USING MORE DISK DRIVES

218 DISKETTE NAMES

219 DISK DRIVE NAMES

220 LESSON MODE
220 Specifying a Diskette Name
222 Specifying a Disk Drive Name

222 AUTHOR MODE

Apple PILOT is generally intended for use with two different Apple computer system configurations: an Author Mode system with two disk drives, complemented by several Lesson Mode systems each using one disk drive. If your systems are set up with these standard configurations, you will not need to know the information in this appendix. In a standard system, Apple PILOT handles all access to the disk drives automatically.

However, it is possible to use Apple PILOT effectively on systems with more than the usual number of disk drives. In particular, Lesson Mode can accomodate up to six disk drives, and can use files from diskettes in all the drives while running a lesson. The information in this appendix is for people using such non-standard, multiple-drive systems.

DISKETTE NAMES

Each time you initialize a new diskette for storing your lesson files, Apple PILOT asks you to

```
ENTER NAME OF DISKETTE, OR
JUST PRESS RETURN TO GIVE IT
THE NAME LESSONS.
```

You can type any name up to seven characters long. If you press the RETURN key without typing a name, your diskette is given the name LESSONS .

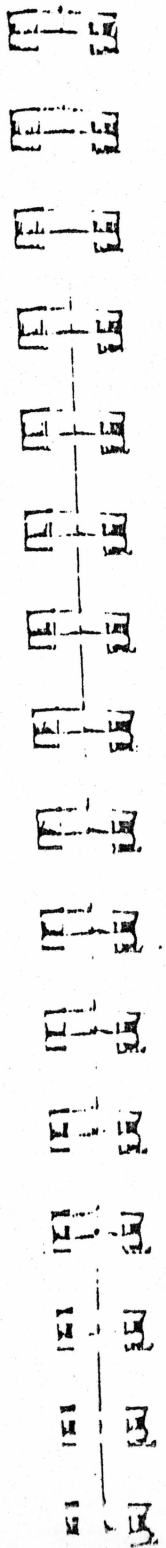
For standard systems (two drives in Author Mode, one drive in Lesson Mode), the name given to lesson diskettes is not very important. This is because you will never have more than one lesson diskette in the system at any time.

***** Warning! *****

In fact, it can be very dangerous to put two diskettes with the same name into the system at the same time. Apple PILOT will be quite confused about which diskette is which. It may retrieve information from the wrong diskette or, worse yet, it may store information on the wrong diskette, possibly even scrambling the directory that tells where all the files are stored.

To avoid this problem, you may wish to develop the habit of giving each new diskette a new and different name. Be sure to write the diskette's name on the label when you initialize it (write very gently, with a felt-tip pen).

If you intend to write lessons that use diskettes from more than one disk drive in Lesson Mode, you MUST give your lesson diskettes different names. If you are not sure they have different names, initialize (and name) a new diskette; then transfer the necessary files from one of the unknown diskettes onto the new one.



[Note: if you have access to an Apple Pascal system, you can use the Pascal Filer to see the name of your lesson diskettes and even to change a lesson diskette's name.]

DISK DRIVE NAMES

Each disk drive has a numeric "name". This name is determined by two things:

1. The number of the slot inside the Apple where the drive's controller card is installed (these slots are numbered from Ø through 7, but PILOT disk controller cards can be in slots 4, 5, or 6).

2. The name of the controller card's connector to which the drive's ribbon cable is attached (the upper connector is labelled "DRIVE 1", the lower connector is labelled "DRIVE 2").

Normally, you don't need to know these names, but for this discussion it will prove helpful. Here are the names for all of the six disk drives you can connect to your Apple:

Name of Drive	Controller Card	Description of the Disk Drive
#4	Slot 6, Drive 1	This is the first disk drive. It is sometimes called "drive 1" or "the primary drive". Every Lesson Mode system must have at least this one drive. In Author Mode, the AUTHOR diskette goes in this disk drive.
#5	Slot 6, Drive 2	This is the second disk drive. It is sometimes called "drive 2" or "the secondary drive." Every Author Mode system must have a lesson diskette in this drive and an AUTHOR diskette in the drive named "#4".
#9	Slot 4, Drive 1	Additional disk drive.
#1Ø	Slot 4, Drive 2	Additional disk drive.
#11	Slot 5, Drive 1	Additional disk drive.
#12	Slot 5, Drive 2	Additional disk drive.

ere is no special reason why the drives have the names they do. The mes may seem arbitrary, but they are correct. If you plan to use a ltiple disk drive Lesson Mode system, you may wish to label your sk drives with their correct drive "names".

ESSON MODE

ssons which are running in Lesson Mode can use files that are stored any diskette in any available disk drive. In general, the safest ractice is to put all the diskettes needed by the lessons into their sk drives BEFORE you turn on the system. That way, when the system tarts itself up it will discover where each diskette is located in the ystem. Thereafter, it is best to leave all the diskettes in their riginal drives. Apple PILOT will look in all the drives to find a iskette whose name was not found during the startup process, but it ssumes the known diskettes will stay put.

n a standard, one-drive Lesson Mode system, you only need to tell the ystem the file name of the lesson, graphic, character-set, or ound-effect which is wanted. Apple PILOT automatically supplies the ame of the only disk drive, #4 .

n a multiple-drive Lesson Mode system, PILOT will continue to supply he name of disk drive #4 for any file that you specify by file name lone. In such a system, however, you now have the option of using a ifferent diskette or a different drive. You can do this in either of wo ways: by giving PILOT the name of the diskette which holds the file ou want, or by giving PILOT the name of the disk drive which holds hat diskette.

SPECIFYING A DISKETTE NAME

This is the preferred method, because it allows your lessons to work correctly no matter which drives the diskettes have been put in. Basically, any instruction which normally specifies a file name alone can also have diskette name appended to the front of that file name, with a colon (:) between the two names. Apple PILOT will then find the diskette with the given diskette name, in any drive, and retrieve the desired file from that diskette.

For example, suppose you have stored a sound-effects file named TUNE1 on a lesson diskette named MYDISK . On a one-drive system, you would put that diskette in the disk drive and run a lesson which would use your sound effect in an instruction such as

```
SX: TUNE                    (Play the sound effect stored
                              in the file named  TUNE , on
                              the diskette now in the disk
                              drive named  #4 .)
```

A multiple-drive system can do the same thing, if that diskette is in the drive named #4 . Suppose, however, that you want your lesson (which might be on the same diskette or on a diskette in a different drive) to use that sound effect even if the diskette containing the sound-effect file is NOT in drive #4 . Just add the diskette's name to the file name in your SX: instruction:

```
SX: MYDISK:TUNE             (Play the sound effect stored
                              in the file named  TUNE , on
                              the diskette named  MYDISK ,
                              in any available disk drive.)
```

In a one-drive system, this instruction still works fine, even though the extra information is not necessary. In a multi-drive system, Apple PILOT will first find the diskette named MYDISK , and then retrieve the file TUNE from that diskette.

The same principle can be used with the instructions GX: , TX: , L: , LX: , FOX: , and FIX: . In fact, when the HELLO program asks you to type the name of the lesson you want to run, you can specify a lesson on any available diskette. Just type the name of the diskette (be sure that diskette is in one of the drives) followed by a colon followed by the name of the lesson.

Here is an example of how your lesson might read a file named TALES , previously stored on a diskette named SNAIL :

```
D: R$(255)                  (Reserve space to store up
                              to 255 characters in string
                              variable  R$ .)
D: Q$(18)                   (Reserve space to store up
                              to 18 characters in string
                              variable  Q$ .)
C: Q$ = "SNAIL:TALES"       (Store the string  SNAIL:TALES
                              in string variable  Q$ .)
FIX: 3Ø;Q$                  (Find the file named  TALES ,
                              on the diskette named SNAIL ,
                              and open it for retrieving
                              records numbered up to 3Ø .)
FI: 12,R$                   (Retrieve the contents of
                              record number 12 , in the
                              currently open file, and
                              store those contents in
                              string variable  R$ .)
T: Tale #12: $R$            (Display text, substituting
                              the tale stored in  R$
                              for  $R$ .)
```

The possibilities of a multiple-drive Lesson Mode system are endless. You could, for instance, have all your graphics files on a special graphics diskette, all your sound-effects on a special sound-effects diskette, and use those diskettes as a "library" of effects that any

lesson could draw on. Or you could write an immense lesson, that ran for hours and hours, Linking from files on one diskette into files on another diskette, and so on.

SPECIFYING A DISK DRIVE NAME

If you wish, you can use the name of the disk drive in place of the diskette's name. For example, if you want to run the sound-effects file TUNE , on the diskette now placed in the disk drive named #9 , you could use this instruction:

```
SX: #9:TUNE          (Play the sound effect stored
                       in the file named  TUNE ,
                       on the diskette now in the
                       disk drive named  #9 .)
```

Apple PILOT will look on whatever diskette is in drive #9 , and retrieve the file named TUNE from that diskette. This method works fine for specifying the name of the lesson you want to run next, in the HELLO program, but is less satisfactory for instructions within a lesson. Every time you run this lesson, you will have to make sure a diskette containing the file TUNE goes in drive #9 . If you put that diskette in drive #1Ø , Apple PILOT won't find it. On the other hand, you could put ANY diskette with a file named TUNE in drive #9 and your lesson would work just fine.

***** WARNING! *****

Specifying a diskette by the name of the drive it is in does NOT let you use two diskettes having the same name. Apple PILOT quickly converts drive names into the name of the diskette found there, so it will still be hopelessly and dangerously confused by finding two diskettes with the same diskette name.

AUTHOR MODE

Author Mode uses only two disk drives, #4 and #5 , even if more drives are connected to the system. The AUTHOR diskette must be in drive #4, and a lesson diskette must be in drive #5 . No matter what name you give a file that you are creating in one of the Editors, that file will always be stored on the lesson diskette in drive #5 . In fact, the Editors do not allow you to type a colon or pound-sign when specifying the name of the file you are creating.

APPENDIX C

ERROR MESSAGES

224 LANGUAGE ERROR MESSAGES

228 SYSTEM EXECUTION ERRORS

LANGUAGE ERROR MESSAGES

When you run a lesson from Author Mode, Apple PILOT gives you various messages to help you locate instructions that are incorrectly written or improperly used, or which have been given numbers that are too big or too small, or in which an expected item is missing. Any time one of these messages is given, the Apple's Error Flag is also raised.

An appropriate error message appears on the screen when Apple PILOT has trouble executing an instruction, temporarily stopping the lesson. Directly beneath the message, you are shown the instruction which contains the error, with the white square cursor located on or just before the part of the instruction where the error was detected. For example, if you try to divide by zero in a Compute expression, you may get a message like this:

```
A-error (Arithmetic error)
c: n = 5 / Ø
```

with the cursor placed on the offending zero. The lesson remains stopped until you press the RETURN key or the space bar. This lets you write yourself a note about the error, in order to correct the problem when you are next in the Lesson Text Editor.

Apple PILOT almost always lets your lesson continue after an error is detected. Usually, the offending instruction (or the offending portion of that instruction) is simply skipped, and execution of your lesson continues at the next instruction. This may have strange effects on your lesson, but you can finish the lesson in some fashion. As you do this, jot down the error message and the offending instruction line. Circle the character which follows the cursor: this is the first character which was found to be unacceptable. When the lesson is over, you can use this information to make appropriate changes to your lesson in the Lesson Text Editor.

Note: these errors messages are not displayed when you run a lesson in Lesson Mode. Instead, the Error Flag is raised and your lesson's error-handling instructions can detect this problem by using the Error Conditioner.

Note 2: if a special character set is in use, the standard character set is invoked before displaying an error message. After the error message, the special character set is NOT automatically re-loaded.

The PILOT Language Error Messages and their explanations follow.

Error Message Displayed	Explanation
A-error (Arithmetic error)	In an expression, you have tried to do something illegal, such as divide by Ø, or use a number that was too big or too small. Most numbers must be in the ranges -3.4E38 to -1.2E-38, Ø , and 1.2E-38 to 3.4E38 . The arguments to some functions must be within smaller limits.
B-error (Value out of bounds)	A subscript number is not in the range specified by Dimension for that variable; or you used a subscripted variable which has not been Dimensioned; or an object-field command number such as a Graphics coordinate is too large (32767 is usually maximum) or is negative.
C-error (Invalid compute instruction)	A bad variable name, bad edit option, missing equals sign, or other syntax error has occurred in the non-expression part of a Compute instruction. Errors in the expression part of a Compute instruction are usually reported as E-errors or S-errors.
D-error (Diskette read error)	Your diskette is damaged (contains bad blocks of information); or a file was read into the Apple incorrectly for some reason, so the end of the current file was reached before the end of the program.
E-error (Invalid expression)	You have written an expression that does not make sense, such as an odd number of parentheses or a wrong character.

Error Message Displayed	Explanation
F-error (File input/output error)	In FOX:, FIX:, FO:, or FI:, you have named a bad string variable; or in FOX: or FIX: the string variable contains a bad file name; or there was no unused space on the lesson diskette for creating a new file; or FO: or FI: used a record number larger than specified by FOX: or FIX: or beyond the file created by FOX:; or you tried to use FO: or FI: when no file was open.
G-error (Invalid graphic instruction)	Something is wrong in the list of commands for a Graphics instruction, such as non-integer number or extra space.
I-error (Invalid function argument)	The argument to a function such as SIN or PDL is too big or too small, or is otherwise incorrect. Such errors are usually reported as A-errors, S-errors, E-errors, or B-errors.
J-error (Invalid jump destination)	A branching instruction such as Jump, Use, or End, or a GOTO command, tried to jump to a label which cannot be found. The offending instruction is not displayed after this message. If branch was to a missing @A, @M, or @P, the system re-starts itself without any error message.
L-error (Invalid link destination)	A Link instruction specified a lesson which cannot be found. The system re-starts itself after this message.
M-error (Invalid modifier/condition)	Where your instruction's list of modifiers and conditioners should be, a character appears that is neither modifier nor conditioner.

Error Message Displayed	Explanation
O-error (Invalid instruction code)	The name of your instruction is not recognized by Apple PILOT.
R-error (Too many arrays or strings)	You have tried to use up more storage places for variables than are available. You can store up to 98 numbers in simple numeric variables and elements of numeric arrays combined. You can store up to 600 characters in reserved string-variable locations and unreserved locations needed when storing a string, combined.
S-error (Invalid syntax or format)	Something is wrong with the way you have typed your instruction, or an expression is too complex to be evaluated.
T-error (Invalid expression type)	You have tried to store a number in a string variable, or a string in a numeric variable, or you have mixed strings and numbers illegally in an expression.
U-error (Too many subroutines)	You have nested Use subroutine invocations more than 4 levels deep. A maximum of four Use instructions can be waiting for their corresponding End instructions at any time.
V-error (Invalid variable name)	The object field of a Compute instruction did not start with a correct variable name, or starts with the name of a string variable you have not Dimensioned; or an Accept instruction used a bad variable name or one not preceded by # or $, or the name of a string variable you have not Dimensioned.
X-error (Invalid execute instruction)	There is something wrong with the string variable named in your XI: instruction.

As you can see, the problems reported by these messages overlap somewhat, and a given message does not always pinpoint a particular problem. Also, the point at which PILOT discovers the problem may not

be the true cause of a problem which may have arisen earlier in the instruction or even many instructions earlier.

The error messages are a useful aid, but you will often need to use some ingenuity to track down what is really causing the reported error. For example, if an error message points out a string constant as causing a T-error, it may not be a problem with the string at all. Instead, you may have forgotten to Dimension a string variable whose contents were to be concatenated to the constant.

Finally, some problems may not cause any error message at all. Instructions, modifier-and-conditioner lists, expressions, and objects which do not follow the rules of Apple PILOT syntax may be executed anyway, with unpredictable results. Only by knowing what should happen in your lessons, and investigating carefully when anything unexpected occurs, will you be able to track down these problems.

SYSTEM EXECUTION ERRORS

When the Apple PILOT system runs into difficulties such as a damaged diskette, a missing file needed to run the next part of the system, or a disk drive door left open, several different things may happen:

1. If you were running a lesson at the time, execution of the lesson will usually just go on to the next instruction. If that is impossible (because you are at the end of the current lesson, Linking to a new portion, say), the current portion of the lesson may be re-started.

2. If the system does not know how to recover, it may re-start itself, returning to the Main Menu (in Author Mode) or to the lesson diskette's HELLO program (in Lesson Mode).

3. Other problems will cause the system to report a

   PILOT EXECUTION ERROR #xxx

   where the number corresponds to one of the following kinds of problems:

Error Number	Description of the Problem
2	The system specified an incorrect disk drive number.
3	The system attempted an illegal operation, such as reading from the printer.
4	(Not used on the Apple.)
5	A disk drive is no longer available, after an operation was successfully started using that drive.
6	A diskette file is no longer in the diskette directory, after successfully starting an operation using that file.
7	An illegal filename was used (for example, a filename more than 10 characters long.)
8	There was insufficient space on the specified diskette to stored another file.
9	The specified diskette was not in any of the disk drives.
10	The specified file was not on the specified diskette.
11	An attempt was made to create a diskette file when a file of that name already exists.
12	An attempt was made to open an already-open file.
13	An attempt was made to access a closed file.
14	There was an error in reading a number (for example, the system expected a number but received a character.)
15	Characters were arriving at the Apple faster than it could handle them.
16	The specified diskette is write-protected.
(Numbers 17 through 63 are not used.)	
64	An attempt to store or retrieve information from a diskette file failed, because the diskette is damaged or the information was stored incorrectly.
(Numbers 65 through 99 are not used.)	
100	A system error of unknown nature has occurred.
101	A number used by the system was too large or too small, or a string was too long.

or ber	Description of the Problem
1Ø2	A diskette file was read into the Apple correctly, but the information in the file was incorrect.
1Ø3	Part of the system program was not available when it was needed, or a program portion was executed incorrectly.
1Ø4	The programs and data in the system have attempted to use up more of Apple's memory than was available.
1Ø5	The system created an integer that was too large.
1Ø6	The system tried to divide something by zero.
1Ø7	Invalid memory reference (not used on the Apple).
1Ø8	The "break" key was pressed (not available on the Apple).
1Ø9	An error occurred in trying to read another part of the system's program into the Apple from diskette.
11Ø	An error occurred when your program tried to store or retrieve information in a diskette file. Usually reported as one of the errors numbered 2 through 64.
111	The system tried to use an instruction that was not available.
112	The system tried to use a number that was too large or too small, or had the wrong format.
113	The system tried to store a string into a string variable that was too small for it.

ter an error which is especially confusing to PILOT, you may have re-start the system yourself by turning the Apple's power off and en on again. All other errors cause the system to re-start self, usually after you press the Apple's spacebar or the RESET key continue.

te that most of these errors occur when the Apple PILOT system is ing its work, especially when you are using the special Editors. ese errors rarely occur when you are running a lesson in Author de, or when a student is running a lesson in Lesson Mode, because ILOT is almost always able to handle any problems that arise when a esson is running.

Execution errors usually result from mistakes in handling the diskettes and disk drives: taking a diskette out of its drive at the wrong time, putting the wrong diskette in a drive, putting a diskette in its drive upside down or sideways, leaving the disk drive's door open, etc. In such cases you can just check your diskettes and disk drives, correct the problem, and re-start the system.

APPENDIX D

PILOT LANGUAGE SUMMARY

234 EXPRESSIONS
234 Numbers
234 Strings
234 Variables
235 Functions
235 Transcendental Functions
235 Arithmetic Functions
236 String Functions
236 Input Functions
236 Operators
237 Order of Evaluating Operations

238 THE PARTS OF AN INSTRUCTION
238 Label
238 Instruction-Name
239 Modifier
239 Conditioners
240 Expression
240 Colon
240 Object
240 Miscellaneous

240 THE PILOT INSTRUCTION SET
240 Text Instructions
240 Response Instructions
241 Control Instructions
241 Computation Instructions
242 Special-Effects Instructions
243 File-Handling Instructions

EXPRESSIONS

NUMBERS

Numbers in Apple PILOT are all "real" numbers, which can be expressed as integers, decimal numbers, or in "scientific" E-type notation:

```
324                    (Integer notation)
45.9716                (Decimal notation)
2.34851E-5             (E-type notation.  This example is
                         2.34851 times 1Ø to the minus fifth
                         power, or  .ØØØØ234851  .)
```

Apple PILOT rounds the main number to six significant digits, if it contains more than six. A number is rounded up if the seventh digit is 5 or more, and down if it is 4 or less; this rounding takes place BEFORE the number is used or stored. Most numeric quantities can be any number in the ranges from -3.4E38 to -1.2E-38 , Ø , and 1.2E-38 to 3.4E38 . Some of the PILOT functions require argument numbers to stay in a smaller range, often from -32767 to 32767 . Most numbers that appear as command parameters in the object field of instructions must stay in the range from Ø through 32767.

STRINGS

Strings are any characters, surrounded by double-quotes:

```
"Apple Computer Inc."
"Today's number is 345.99887, up 3/8ths."
```

VARIABLES

Many instructions may use variables in the object field. A "simple numeric variable" is a number-storage place which holds one number. You give it a name such as F or Q3 . Created by using Compute or Accept to store a number: C:F=15 or A:#Q3 . In a Type or Accept instruction, name must be preceded by # and followed by a space or by end of instruction: T:#F or A:#Q3 .

A "numeric array variable" is a set of number-storage places with one name. Subscripts (one number or two, in parentheses) identify individual elements in the array. You give the array a name such as M or P8 , and elements have names such as M(3,4) or P8(13) . Created by using Dimension to reserve enough space for the intended elements: D:M(5,1Ø) or D:P8(2Ø) .

A "string variable" is a word-storage place which holds a string of characters. You give it a name such as R$ or U5$. Created by using Dimension to reserve enough space for the intended strings (1 to 255 characters): D:R$(3Ø) . In a Type or Accept instruction, name must be preceded by $ and followed by a space or by end of instruction: T:R or A:$U5$.

You can also use subscripts after a string variable name to identify one character (character-number) or a substring of contiguous characters (first-character-number,number-of-characters) to make "substring variables" with names such as R$(23) or U5$(7,22) .

A "system variable" is a number-storage or word-storage place created and used by Apple PILOT. You do not store things in these variables, but you can use what PILOT puts in them. There are four system variables, with the following names and contents:

%A	Answer Count	(number of times in a row the last Accept instruction has been executed),
%B	Answer Buffer	(the last Accepted typed response),
%X	X-Coordinate	(graphics screen coordinates of the last
%Y	Y-Coordinate	point accepted by an AP: instruction).

There are 98 number-storage places for all simple numeric variables and numeric array variable elements. There are 6ØØ character-storage places shared between places reserved by Dimensioning string variables and unreserved places needed for temporarily holding any string before storing it in a string variable.

FUNCTIONS

A function is said to "return" the result of its calculation or operation. The information in parentheses after the function's name is called the function's "argument" or arguments. A numeric argument (shown as X) may be a number, a simple numeric variable, an element of a numeric array variable, or an expression whose result is a number. Except as noted in the descriptions, the value of any numeric argument can be any valid real number. A string argument (shown as X$) may be a quoted string, a string variable, a substring variable or an expression whose result is a string.

Transcendental Functions

SIN(X) Returns sine of angle X . Angle X expressed in radians. If X exceeds range -1Ø2942 through 1Ø2942, SIN(X) returns Ø.

COS(X) Returns cosine of angle X . Angle X expressed in radians. If X exceeds range -1Ø294Ø through 1Ø294Ø, COS(X) returns Ø.

ATN(X) Returns the radian measure of an angle whose tangent is X . If X exceeds range -5.3E18 through 5.3E18, ATN(X) returns Ø.

Arithmetic Functions

SGN(X) Returns -1 if X is negative, 1 if X is positive, or Ø if X equals zero.

ABS(X) Returns the absolute value of X .

FIX(X) Returns integer found by truncating X (truncation discards any part of X to right of decimal, without rounding). X must be in range from -32767.9 through 32767.9 .

INT(X) Returns largest integer less than or equal to X . X must be in range from -32766.9 through 32767.9 .

RND(X) For X less than 1, returns random real number in the range from Ø through .999969 . For X from 1 to 32767.9, returns random integer from Ø through FIX(X)-1 .

SQR(X) Returns the square root of X . X must be greater than zero.

LOG(X) Returns base 1Ø logarithm of X . X must be greater than zero.

LN(X) Returns natural logarithm of X . X must be greater than zero.

EXP(X) Returns e to the X power. For X greater than 89.4, EXP(X) returns the value Ø . e is approximately 2.71828 .

String Functions

ASC(X$) Returns the decimal ASCII code for the first character in X$.

CHR(X) Returns the character whose decimal ASCII code is FIX(X) . X is normally in the range from Ø through 127 .

FLO(X$) Returns the value of the first number found in string X$.

STR(X) Returns the number in X expressed as a string of characters.

INS(N,T$,P$) Starting at character position FIX(N), searches through target string T$ for the first occurrence of pattern string P$. Returns Ø if no occurrence found, or returns the position in T$ of the matching sequence's first character. N and T$ may not be expressions.

LEN(X$) Returns the number of characters currently stored in X$.

Input Functions

PDL(X) Returns an integer from Ø through 255 indicating the position of the knob on game control ("paddle") number X . First two game controls are usually numbered Ø and 1 .

BTN(X) Returns 1 if the button on game control number X is being held down, or returns Ø if that button is not being pressed.

KEY(X) Returns 1 if any keyboard key has been pressed since the last Accept or Wait instruction ended, or returns Ø if no key has been pressed. The value of X is not used.

TIM(X) Returns the number of seconds the student took to respond to the last Accept or Accept Point instruction. If Accept was terminated by PR:Tn timeout, TIM(X) returns Ø . The value of X is not used.

OPERATORS

Most operators are placed between the two elements which are used in carrying out that operation. A numeric element (shown as X or Y) may be a number, a simple numeric variable, an element of a numeric array variable, or an expression whose result is a number. Except as noted in the descriptions, the value of any numeric element can be any valid real number. A string argument (shown as X$ or Y$) may be a quoted string, a string variable, a substring variable or an expression whose result is a string.

The standard operators are listed in the precedence table, which follows. This section discusses only items which might not be obvious.

X ** Y The result of this operation is X raised to the power Y . If the result would exceed the range -6.8E38 to 6.8E38, the result is Ø . If Y is zero, result is 1 , except if X is zero the result is always Ø . If X is negative, Y must be an integer in the range -32767 to 32767, and result is positiv for even Y , negative for odd Y .

X = Y
X > Y
etc. Relational comparisons form an "assertion", such as "X is equa to Y". When X and Y are evaluated, if the assertion is true, the result is 1; if the assertion is false, result is Ø

X$ = Y$
X$ > Y$
etc. String assertions compare ASCII codes of the first character in each string. If the two numbers are different, assertion is evaluated for those two numbers as true (result 1) or false (result Ø). If the two first numbers are equal, the second character in each string is compared the same way, and so on. The two strings are made equal in length by adding spaces (ASCII 32) to the shorter one. Two strings are considered equal only if all the character-pairs compared are equal.

NOT X
X AND Y
X OR Y The result of NOT X (or ~X) is 1 if X is Ø , or the result is Ø if X is non-zero. X AND Y (or X&Y) is an assertion that "X is non-zero AND Y is non-zero". X OR Y (or X!Y) is an assertion that "X is non-zero OR Y is non-zero, or both ar non-zero". If the assertion is true, the result is 1 ; if the assertion is false, the result is Ø .

X$!! Y$ The concatenation operator joins string Y$ to the end of string X$, making one string out of two.

ORDER OF EVALUATING OPERATIONS

Here is the order in which the various operations are carried out in an expression:

1. All variables are replaced by the values stored in them.
2. Functions are evaluated (this may require evaluating an expression which appears as an argument).
3. Operations within the innermost set of parentheses are evaluate before proceeding to the surrounding expression.
4. When the order of the operations in any part of the expression not set by parentheses, the remaining operations are carried ou in the following sequence:

First	NOT	Logical NOT	NOT X
	~	Logical NOT	~X
	-	Unary Minus Sign	-X
	+	Unary Plus Sign	+X

Second	**	Exponentiation	X ** Y
Third	*	Multiplication	X * Y
	/	Division	X / Y
Fourth	+	Addition	X + Y
	-	Subtraction	X - Y
	||	Concatenation	X$ || Y$

Fifth	=	: Equal	X = Y	X$ = Y$
	<>	: Not Equal	X <> Y	X$ <> Y$
	<	Relational : Less Than	X < Y	X$ < Y$
	>	Comparisons : Greater Than	X > Y	X$ > Y$
	<=	: Less Than or Equal	X <= Y	X$ <= Y$
	>=	: Greater Than or Equal	X >= Y	X$ >= Y$

Sixth	AND	Logical AND	X AND Y
	&	Logical AND	X & Y
	OR	Logical OR	X OR Y
	!	Logical OR	X ! Y

Operations of equal precedence are executed from left to right. There can be up to nine operations "on hold", awaiting the outcome of another operation before they can be evaluated. Open parentheses are counted as a waiting operation.

THE PARTS OF AN INSTRUCTION

The parts of an Apple PILOT instruction should appear in the following order [items in square brackets are optional]:

[*label] instruction-name[modifiers&conditioners][(expression)]: [object]

LABEL

Any name up to six characters long identifying an instruction or lesson section. Where the identifying label is placed in the lesson, it is preceded by an asterisk (*) and separated by a RETURN or space from the following instruction-name. A branching instruction can then use the label name (without the *) in its object field as the destination of a jump.

INSTRUCTION-NAME

Required except when continuing instruction from previous line. The one, two, or three-letter instruction names, and the identifying word we can associate with them, are:

R	Remark	J	Jump	D	Dimension	G	Graphics
T	Type	U	Use	C	Compute	GX	eXecute Graphics
		E	End				
PR	PRoblem	L	Link	FOX	Open New File	S	Sound
A	Accept	XI	eXecute Indirect	FIX	Open Old File	SX	eXecute Sound
M	Match			FO	File Output		
		W	Wait	FI	File Input	TX	eXecute Char-Set

MODIFIER

One-letter element, in list of modifiers and conditioners which follows the instruction name and precedes the expression (if any) and the colon. Each modifier has a specific effect on one instruction. Used with other instructions, that modifier has a different effect or is ignored.

TH:	(Hang modifies Type)	No carriage-return after a Typed line.
AX:	(eXact modifies Accept)	Accept exact response, with no editing.
AS:	(Single modifies Accept)	Accept a single-character response.
AP:	(Point modifies Accept)	Accept an X,Y point from game controls.
MS:	(Spell modifies Match)	Allow one-letter misspelling on a Match
MJ:	(Jump modifies Match)	Jump to next Match if this Match is unsuccessful.
LX:	(Erase modifies Link)	Start a new lesson without preserving the old lesson's variables.

In addition to these, the X (eXecute) modifier is used with five more instructions (GX:, SX:, TX:, FIX:, and FOX:). These instructions are changed so much that they are discussed as separate instructions.

CONDITIONERS

One-letter or one-digit element, in list of conditioners and modifiers which follows the instruction name and precedes the expression (if any) and the colon. Each conditioner establishes a test or condition for use with any instruction. Instruction is skipped unless each named condition is met. Up to four may be used effectively, in any order, o from each type:

Y	(Yes conditioner)	Execute if last Match was successful.
N	(No conditioner)	Execute if last Match was unsuccessful.
1 to 9	(Answer-Count conditioner)	Execute if the conditioner number equals the current Answer Count.
E	(Error conditioner)	Execute if the Error Flag has been raise
C	(Last-expression conditioner)	Execute if last evaluated instruction-modifying expression was True (non-zer

EXPRESSION

One arithmetic or "assertion" expression, in parentheses. One may be used with any instruction. Instruction is skipped unless the evaluated assertion is true, or the evaluated expression is non-zero.

Arithmetic Expression:	(T-13)	Execute if result is non-zero.
Assertion Expression:	(T>13)	Execute if assertion is true.

COLON

Every instruction line must contain a colon (:) after the instruction name, modifier and conditioner list (if any), and expression (if any). Colon as the first character on a line continues the previous instruction up to 25Ø characters total.

OBJECT

Depends on the instruction. Preceding spaces are allowed. Type instruction displays those spaces; other instructions do not.

MISCELLANEOUS

Every instruction, including the last instruction in a lesson, must end with a RETURN character. Instructions may be typed in any mixture of upper and lower case letters, with these exceptions: the object text of both Type and Match will be used exactly as typed. Blank lines are allowed between different instructions, and are ignored. Only the first 25Ø characters of an instruction (not counting continuation-colons) are used when the instruction is executed.

THE PILOT INSTRUCTION SET

TEXT INSTRUCTIONS-CHAPTER 2

Remark	R: THIS IS A COMMENT.	Comments to author; not executed.
Type	T: What is an apple?	Displays object text on the screen.
	T: #N is right, S .	Text may include variable-names.

RESPONSE INSTRUCTIONS-CHAPTER 3

Problem	PR: E G T2Ø	Starts new section and sets options (if any option is set, any options not mentioned are turned off):
Escape	PR: E	If any response begins with @ , Uses subroutine labelled SYSX .
GOTO	PR: G	If any response begins with GOTO , uses remainder as object for Jump.
Lower-case	PR: L	Converts responses to small letters.
Upper-case	PR: U	Converts responses to capital letters.
Spaces	PR: S	Removes all spaces from responses.
Time	PR: Tn	Sets a maximum response time of n seconds (TØ-default, unlimited time
Wipe-labels	PR: W	"Forgets" all previous labels, from this point forward.
	PR:	Starts new section without changing any previously selected options.
Accept	A:	Accepts student's response. May
	A: #N L	assign response to variable(s).
Match	M: COMPUTER	Looks for object text in student's
	M: APPLE|BANANA	last Accepted response.

CONTROL INSTRUCTIONS-CHAPTER 4

Jump	J: NEXT	Branches to label, next PR: (@P),
	J: @M	next M: (@M), or last A: (@A).
Use	U: SCORE	Branches like J:, usually to labelle
	U: @P	subroutine which ends with E: . "Remembers" place to return to.
End	E:	Returns to instruction following U:
	E: @A	that invoked subroutine, or retur and then branches like J: . If n reached through U: , ends lesson.
Link	L: LESSON2	Starts new lesson; keeps old variab
	L: LESSON2,REVIEW	May start at a specified label.
eXecute Indirect	XI: Q2$	Executes contents of string variabl as an Apple PILOT instruction.
Wait	W: 5	Pauses the specified number of secc or until any key is pressed.

COMPUTATION INSTRUCTIONS-CHAPTER 5

Dimension	D: Q2$(4Ø)	Sets maximum string size for strin;
	D: R(4,5)	variable or maximum subscript fo numeric array variable.
Compute	C: Q2$ = "APPLE"	Evaluates an arithmetic or string
	C: M = 417.95	expression, and stores it in the
	C: H3 = COS(FLO(A$)-2)	variable to left of equals sign.
	C: /Q2$ /@/ C	Edits contents of specified string variable, with these options:
Upper-Case	C:/S$ U	Converts all letters to capit:
Capitalize	C:/S$ C	Capitalizes first character, if it is a letter.

Delete	C:/S$ /c/	Removes every character c .
Replace	C:/S$ /xy	Replaces every character x with the character y .

SPECIAL EFFECTS INSTRUCTIONS–CHAPTER 6

Graphics	G: C6;P23,4Ø;D98,13	Reads graphics routines from diskette and executes the graphics commands which appear in the object field:
Viewport	G: Vl,r,t,b	Sets left, right, top, and bottom character positions for text viewport. Default is full screen. Columns = Ø to 39 (left to right), and rows = Ø to 23 (top to bottom).
	G: V	Re-sets full screen text viewport.
Erase Screen	G: ES	Erases the text viewport to black without using graphics routines.
	G: ESn	Erases screen to color n (Ø to 9) without changing drawing color. Ø = Black1 4 = Black2 8 = Reverse 1 = Green 5 = Orange 9 = White 2 = Violet 6 = Blue (1-dot) 3 = White1 7 = White2
Color	G: Cn	Sets drawing color to n (Ø to 9).
Offset	G: Ox,y	Defines absolute x,y as reference point for relative locations, and moves pen there. Graphics x = Ø to 56Ø (left to right), y = Ø to 511 (bottom to top).
Move	G: Mx,y	Moves pen to point at relative x,y .
Plot	G: Px,y	Plots point at relative x,y .
Quit	G: Qx,y	Plots black point at relative x,y .
Draw	G: Dx,y	Draws line to relative x,y .
Redraw	G: Rx,y	Draws black line to relative x,y.
Type	G: Ttext	Places text on screen starting at current graphics cursor position.
Xmit	G: Xn1,n2,n3	Transmits screen-control characters whose ASCII codes are n1,n2,n3.
eXecute Graphics	GX: MAP	Draws image in named file, step by step.
	GX: PHOTO2I	Quickly replaces the screen with complete image in named file.
Sound	S: 2,12;4,24;3,12	Plays sounds of pitch,duration . Pitch = 1 (low) to 5Ø (high) Pitch = Ø is silent (rest) Duration = Ø (short) to 255 (long)
eXecute Sound	SX: TUNEUP	Plays sound effect stored in named diskette file.
eXecute char-set	TX: CYRILLIC	Starts using characters in named file.
	TX:	Goes back to standard character set.

FILE HANDLING INSTRUCTIONS–CHAPTER 7

Open new file	FOX: n,X$	Creates and opens new file with name stored in variable X$, to contain records Ø through n .
	FOX: Ø,X$	Finds existing file with name stored in variable X$, and deletes it from the diskette.
Open existing file	FIX: n,X$	Opens existing file with name stored in variable X$, for reading or writing records Ø through n .
File Output	FO: n,X$	Stores contents of variable X$ in record n of the file now open.
File Input	FI: n,X$	Reads record n of the file now open, and stores result in variable X$.

Note: L:, LX:, GX:, SX:, TX:, FIX:, and FOX: all close any file already open, before attempting to open a new file.

INDEX

A

A: Accept instruction 48, 122, 156, 159, 162, 166
  answer counting 51, 169
  modifiers 54, 122
  response editing 49
  response timing 49, 122, 200
  unlabelled destination 55
  variables 52, 166, 168
Accept Point instruction 122, 199, 201
addition operator 202
ampersand 59, 205
AND logical operator 205
"and" Match controller character 59
angle functions 189
Answer Buffer (system variable %B) 26, 52, 156, 186
Answer Count (system variable %A) 26, 51, 157, 186
Answer-Count conditioner 13, 14, 186
answer counting 51, 169
Apple Pascal see Pascal
Apple PILOT, differences 2, 7, 137
arctangent function 189
arguments of functions 188, 206
arithmetic expressions 206
arithmetic functions 191
arithmetic operators 202
array, numeric 89, 177
  see numeric array variable
array, string 182
ASCII characters and codes 127, 203, 209
  functions 193
ASC(X$) character-to-code function 193
assertion 22, 203
assignment, in Compute 93
asterisk 6, 7, 59, 205
ATN(X) arctangent function 189
Author Mode 28, 54, 119, 218, 222, 224

B

backspace 108, 211
bell or beep 211
blank lines 28
blank Remark lines 32
blank Type lines 35
blocks, diskette 136, 140
BTN(X) game button function 199
Buffer, Student Answer (system variable %B) 186
button, game control 122, 199

C

C Capitalize option, Compute 102
C Color command, Graphics 106
C Last-expression conditioner 13, 20
C: Compute instruction 93
CAI 2
calculation expression 22
calculations 93, 173
  in SYSX Escape routine 161
capital letters 4, 27, 103, 139, 143
Capitalize option, Compute 102
capitalize option, Accept 49
  Problem 45
character codes, ASCII 193, 209
Character Set Editor 126
character sets 107, 126, 215
  creating 127
  files 126
  using 127
character variables 182
  see string variables
CHR(X) code-to-character function 193
clear screen 106, 211
clear to end of line 211
closing a file 137
colon 3, 10, 23
Color command, Graphics 106
color in graphics 111, 115
comments 32
Common PILOT, variations from 2, 7, 137
compacting a diskette 140

comparisons of numbers or strings 203
Compute instruction 93, 95, 167, 169
used for string editing 101, 172
Computer Assisted Instruction 2
concatenation operator 205
conditioner 6, 13, 239
constant 94, 174
continuation-colon 5, 24, 35, 58, 110, 130, 170
control characters 108, 210
control options, PRoblem 44
controller characters, Match 59
controls, game 122, 199
conversion, ASCII to character 108, 193, 210
character to ASCII code 193, 210
real number to integer 191
number to string 196
string to number 53, 194
to non-standard character set 126
coordinates, Graphics 111
Accepting 122
COS(X) cosine function 189
counting answers 51, 169, 184, 186
Create and Open New File instruction 137, 138
creating variables 26, 88, 95
character sets 127
data files 140
graphics images 119
numeric array variables 89, 178
simple numeric variables 175
sound effects 132
string variables 88, 180
cross-hairs, Accept Point 122
CTRL-A 3, 27
CTRL-C 30, 137, 162
CTRL-Z 3, 27
cursor 210

D

D Draw command, Graphics 107
D: Dimension instruction 88
.DATA file name suffix 136
default Graphics color 116
delete option, Compute 102
deleting data files 139
de-bugging messages 224
Dimension instruction 88, 166, 178
with Link 78
dimensions of numeric array variables 89, 177
disk drives, multiple 217
names 219
diskette blocks 140
diskette files, character set 126
data 135
graphics 119
sound effects 132
diskette names 218
display characters 213, 215
display modes 213
division 202
DLE indent code 211
dollar sign, with string variable names 26, 37, 52, 88, 166, 180
Draw command, Graphics 107
drawing color, Graphics 106, 116
duration, note 129

E

E Error conditioner 16
E Escape option, PRoblem 45, 159
E: End instruction 74
editing a string with Compute 101
of Accepted responses 44, 49
of Match text 60
Editor, Character-Set 126
Graphics 119
Sound Effects 132
also, see Lesson Text Editor
elements in expressions 207
of instructions 7
of numeric array variables 89, 95, 177
End instruction 74
equal operator 204
Erase modifier 12, 77
Erase-Screen command, Graphics 106
Error conditioner 13, 16
Error flag 16, 21, 53, 119
error messages 29, 223
error-handling 10, 18, 150, 156, 159
ES Erase-Screen command, Graphics 106
Escape command 30, 159
Escape option, PRoblem 45, 46, 159
escaping from a program loop, by Answer Count 170
by CTRL-C 162
by Escape 159
by GOTO 156
by RESET 163
evaluation of expressions 206-208
eXact modifier 12, 54
exclamation point 59, 119, 205, 206
exclusive-OR display mode 214
eXecute character-seT file instruction 126
Graphics file instruction 119
Indirect instruction 81
modifier 12
Sound file instruction 132
execution order, in expressions 97, 206
execution-time commands 29, 46, 155
exponentiation 193, 202
expressions 6, 21, 96, 206
EXP(X) e-to-the-X function 193

F

false (value is 0) 203
FI: Input from open File instruction 151
File Input instruction 151
file names 119, 120, 136, 139
File Output instruction 146
files, character-set 126
data 135
graphics 119
lesson 28, 77, 217
sound 132
FIX: Open Existing File instruction 142
FIX(X) truncation function 191
FLO(X$) string-to-number function 194
form of instructions 7, 236
form of operations and expressions 202-208
formatting screen displays 34, 112
FO: Output to open File instruction 146
.FOTO file name suffix 120, 136
FOX: Create and Open New File instruction 138
full screen (no viewport) graphics 113
functions 188-201
arithmetic 191
input 199
string 193
summary 235
transcendental 189

G

G GOTO option, PRoblem 156
G: Graphics instruction 106
game controls 122, 186, 199
GOTO execution-time command 30, 156
GOTO option, PRoblem 45, 46, 156
Goto XY 213
.GRAF file name suffix 120, 136
graphics coordinates, Accepting 122
graphics cursor 113, 114, 118
Graphics Editor 119
graphics files 119
Graphics instruction 106
colors 115
commands 106
command rules 108
cursors 113, 114, 118
screens 110
graphics screen 110
Hi-Res 111
PILOT 110
greater-than operator 204
greater-than-or-equal operator 204
GX: eXecute Graphics instructio 119

H

H Hang modifier 12, 34, 39
HELLO lesson 29, 78
Hi-Res Graphics screen 111
home cursor 213
Horizontal Coordinate (system variable XX) 186

I

indent 211
initializing a diskette 218
Input from open File instruction 151
input functions 199
instruction format 3, 7, 238
  length 27
  line 23
  names 10, 239
  object 25
  summary 238
instruction-modifying expression 20, 21
INS(N,T$,P$) word-in-string function 197
INT(X) real-to-integer function 191
inverse video 3, 211

J

J Jump modifier 12
J: Jump instruction 66
Jump instruction 66
Jump modifier 12

K

KEY(X) keypress function 200

L

L Lower-case option, PRoblem 44
L: Link instruction 77
labels 7, 46, 77, 156, 238
language error messages 224
Last-expression conditioner 13, 20
length of instructions 27
LEN(X$) length-of-string function 83, 198
lesson diskette 77, 119, 126, 132, 136, 139, 217
Lesson Mode 29, 140, 143, 218, 220, 224
Lesson Text Editor 4, 27, 32, 34, 58, 83, 109, 117, 130
less-than operator 204
less-than-or-equal operator 204
limits, Accepted numbers 53
  Accepted strings 186
  array-element subscripts 89
  character positions 108, 109, 112
  data file size 139
  expressions 208
  functions 188-201
  graphics coordinates 107, 109, 110, 113
  numeric variables 89, 176
  operators 202-206
  real numbers 174
  record numbers 139, 143, 146, 151
  response time 44, 49, 123
  string variables 88, 181
  subscripts 178, 183
  variable names 175
line drawing 107
linefeed 211
Link instruction 77
LN(X) natural logarithm function 192
logical operators 204
LOG(X) base-10 logarithm function 192
loop 163
  see escape from program loop
lower case letters 4, 27, 103, 139, 143
Lower-case option, PRoblem 44

M

M Move command, Graphics 107
M: Match instruction 58
Match instruction 3, 58, 158
  editing 60
Match word 58, 172
maxima see limits
modes, screen 213
modifiers 6, 11, 239
  with Accept 54
  with Graphics 116
  with Link 77
  with Match 60
  with Sound 130
  with Type 39
Move command, Graphics 107
multiple disk drives 217
multiplication 202
music 129

N

N No conditioner 13
names of instructions 10, 238
  of disk drives 219
  of diskettes 218
  of variables 175, 177, 180, 183
natural logarithm function LN(X) 192
negative numbers 201
nesting expressions 207
  subroutines 74, 76
new data files 137, 139
No conditioner 13
non-printing control characters 108, 210
normal display mode 213
normal video 211
NOT logical operator 205
notes, musical 129
not-equal operator 204
null characters 139
number comparisons 203
numbers 174, 234
  in Accepted responses 53, 104
  in strings 53, 194
  notation 172, 174
number-storage place 26, 166
number-to-string function FLO(X$) 196
numeric array variable 26, 89, 93, 95, 177
numeric variable 26, 156, 168
  see simple numeric variable and numeric array variable

O

O Offset command, Graphics 107
object 5, 25
Offset command, Graphics 107
Open Existing File instruction 137, 142, 151
Open New File instruction 138
Open Old File instruction 142
opening a data file 136, 138, 142
operators 202-206
  logical 204
  precedence 97, 206
  relational 203
  summary 236
  string 205
OR logical operator 205
"or" Match controller character 59, 205
order for evaluating operations 206
Output to open File instruction 146
overprint display mode 214

P

P Plot command, Graphics 107
P Point modifier, Accept 122
paddles
  see game controls
parentheses, in expressions 97, 206-208
  with subscripts 89, 177, 182
parts of an instruction 7
Pascal, filer 120, 136
PDL(X) game control function 199
percent sign 59
PILOT language 2
  Execution Error messages 228
pitch, note 129

Plot command, Graphics 107
plotting, reverse color 124
plus sign 202
Point modifier 12, 55, 122
point plotting 107
pound sign 38, 52, 168
precedence among operations 97, 206, 237
precision of numbers 174
PR: PRoblem instruction 44
PRoblem instruction 44, 158, 162, 172
  as unlabelled branch destination 9, 44, 47
program 3
program segments, Linking 77
pseudo-variables 184

Q

Q Quit command, Graphics 107
quick-draw graphics files 119
Quit command, Graphics 107

R

R Redraw command, Graphics 107
R: Remark instruction 32
random number function RND(X) 192
real numbers, limits 54
records, number 136, 137, 146, 151
Redraw command, Graphics 107
relational operators 203
relative branch destinations 9
relative graphics coordinates 107, 109, 110
Remark instruction 32
removing data files 139
replace option, Compute 102
RESET 30, 137, 163
Response Buffer 185
  see Student Response Buffer
response editing 45, 49, 101
response timing 45, 49, 122, 200
response-timing option, PRoblem 45
retrieving
  stored character sets 126
  stored data records 136, 151
  stored graphics images 119
  stored sound effects 132
RETURN 27, 58, 210
return-address stack, Use 69, 76
Reverse, Graphics color 115, 116
re-open existing file 138
RND(X) random number function 192
rounding of numbers 174
running a lesson 28
run-time commands 155

S

S Single modifier 12, 55
S Spaces option, PRoblem 44
S Spelling modifier 12, 60
S: Sound instruction 129
saving
  see storing
scientific notation 54, 104, 172, 174
screen, command characters 210
  coordinates, Accepting 122
  graphics 110-111
  modes 213
  text 34, 112
segments, Linking 77
SGN(X) sign function 191
sign function 191
significant digits 174
simple numeric variable 26, 84, 95, 111, 168, 175
Single modifier 12, 55
SIN(X) sine function 189
.SONG file name suffix 136
sound effect 130
Sound Effects Editor 132
sound effects files 132
Sound instruction 129
spaces, in Accepted responses 44, 50
  allowed in instructions 7, 10, 13, 21, 23, 25, 38, 52, 109, 130, 167, 169
  in Type text 36, 38
Spaces option, PRoblem 44
speed, execution 100
Spelling modifier 12, 60
SQR(X) square root function 192
square root function 192
stack, subroutine return 76
standard PILOT, variation from 2, 7, 137
storing character sets 126
  graphics images 119
  numbers 52, 136, 93, 168, 174
  screen positions 122
  sound effects 129
  strings 52, 93, 136, 166, 180
  data records 136, 146
  lesson data 135
  student responses 52, 122, 147, 186
string 174, 234
  comparisons 203
  expressions 206
  functions 193
  concatenation operator 205
string variables 26, 88, 95, 166, 180
  in file instructions 136
  editing 101
  execution as instruction 81
string-to-number function 194
STR(X) number-to-string function 196
Student Answer Buffer (system variable %B) 26, 52, 186
Student Response Buffer (system variable %B)
  see Student Answer Buffer
student responses 48, 122, 155, 186
student's name, using 166
student's numbers, using 168
subroutine 69, 74, 76
subscripts 26, 177, 183
substring variables 182
subtraction 202
sub-expressions 97, 207
summary of PILOT language 233
SX: eXecute Sound file instruction 132
system error messages 228
system variables 26, 185
  Answer Count %A 51
  Student Answer Buffer %B 52
  X-Coordinate %X 122
  Y-Coordinate %Y 122
SYSX Escape subroutine 159

T

T Time option, PRoblem 44, 45, 49, 85, 122
T Type command, Graphics 107
T: Type instruction 38
testing a lesson 28, 157, 224
text characters, alternate sets 126
  ASCII codes 215
  in graphics displays 100, 107, 111, 112
  in text displays 34
text cursor 34, 39, 113, 118, 210
text displays 34, 107, 112
Text Editor 3
  see Lesson Text Editor
text screen 112
text viewport 36, 112, 114, 211-212
  see viewport
text window 36, 112, 114
  see viewport
TIM function 49, 122, 200
Time option, PRoblem 44, 45, 49, 123
timed Wait 85
timing of responses 44, 45, 49, 85, 122
TIM(X) response time function 200
transcendental functions 189
trigonometric functions 189
true (value is not 0) 203
truncation function 191
TX: eXecute character-seT file instruction 119
Type command, Graphics 107
Type instruction 34
  modifiers 39
  variables 37

U

U Upper-case option, Compute 102
U Upper-case option, PRoblem 44, 50
U: Use instruction 69
unary plus and minus operators 202

unequal operator 204
unlabelled destinations 9, 66, 70, 75, 156
  Accept 55
  Match 61
  PRoblem 47
unused diskette blocks 140
upper and lower case letters 4, 27, 103, 139, 143
Upper-case option, Compute 102
Upper-case option, PRoblem 44
Use instruction 69, 159

V

V Viewport command, Graphics 108
variables 26, 94, 100, 166, 175, 234
  in Accept 52, 175, 179, 180
  in Compute 93, 101
  in Dimension 88
  in eXecute Indirect 81
  in file instructions 136
    FIX: 143
    FI: 151
    FOX: 139
    FO: 146
  in Graphics 109, 111
  in PRoblem 46
  in Sound 129
  in subscripts 177, 183
  in Type 37, 175, 179, 180
  in Wait 84
  passed by Linking 77
Vertical Coordinate (system variable %Y) 186
video, normal and inverse 27, 211
viewport 36, 112, 114, 118, 120, 211-212
Viewport command, Graphics 108

W

W Wipe-labels option, PRoblem 9, 45, 46
W: Wait instruction 84
Wait instruction 84
wild-card character, Match 59
window, text 36, 112
  see viewport
Wipe-labels option, PRoblem 9, 45, 46
word-in-string function 197
word-storage place 26, 166

X

X Erase modifier 12, 77
X eXact modifier 12, 54
X eXecute modifier 12
X Xmit command, Graphics 108, 209
XI: eXecute Indirect instruction 81
Xmit command, Graphics 108, 209
X-Coordinate (system variable %X) 26, 122, 186

Y

Y Yes conditioner 13
Y-Coordinate (system variable %Y) 26, 122, 186

Z

SYMBOLS

! logical OR operator 205
  Match "or" controller 59
|| concatenation operator 205
" around string constants 174
# on simple numeric variable names 26, 37, 52, 168, 175
$ on string variable names 26, 37, 52, 88, 166, 180
% Match "start-word" controller 59
& logical AND operator 205
  Match "and" controller 59
() parentheses, in expressions 97, 206
  in subscripts 88, 177, 182
* Match "wild-card" controller 59
  multiplication operator 202
** exponentiation operator 202
+ addition operator 202
  unary plus sign 202
, between object elements 77, 135
  between elements in a command 106, 129
  between numbers in a subscript 88, 177, 182
- subtraction operator 202
  unary minus sign 202
/ Compute string-editing symbol 101-102
  division operator 202
: before object of instruction 23
  to continue an instruction 24
  in file names 220
; between commands in a list 106, 129
< less-than operator 204
<= less-than-or-equal operator 204
<> unequal operator 204
= equal operator 204
> greater-than operator 204
>= greater-than-or-equal operator 204
@ Escape command 159
^ logical NOT operator 205
%A Answer Count 51, 186
%B Student Answer Buffer 52, 186
%X X-Coordinate 122, 186
%Y Y-Coordinate 122, 186